IBM

Personal Computer Hardware Reference Library

BASIC Reference

6361134

International Business Machines Corporation

P.O. Box 1328-C
Boca Raton, Florida 33432

6361134

Printed in the United States of America

IBM

Personal Computer
Hardware Reference
Library

BASIC Reference

(Third edition May, 1984)

Preface

The IBM Personal Computer offers three upwardly compatible versions of the BASIC interpreter: Cassette, Disk, and Advanced. This *BASIC Reference* and the companion volume, the *BASIC Handbook*, are references for these three versions of BASIC—and for the various releases of BASIC, beginning with 1.0.

It is important for you to know that these books are written *only as references* for the BASIC programming language, not as textbooks that will teach you how to program. If you need step-by-step instruction in learning to program in BASIC, we suggest that you ask for the materials you need at your library, bookstore, or computer store.

This book is an encyclopedia-type manual. It contains, in alphabetical order, the syntax and semantics of every command, statement, and function in BASIC.

The *BASIC Handbook* contains general information about using BASIC. There are sections that will help you get started using BASIC, and there are some sections that contain information on advanced subjects for the experienced programmer.

These two BASIC books are extensively cross-referenced and indexed. Each also includes appendixes of useful information. In addition, there is a Quick Reference that lists all the BASIC commands, statements, and functions, categorized by task.

> **Note:** If you have an IBM PC*jr*, use the *BASIC* book specifically for the PC*jr* rather than these books.
>
> The IBM Personal Computer BASIC Compiler is an optional software package that is available at your

computer store. If you have the BASIC Compiler, use the IBM Personal Computer *BASIC Compiler* book in conjunction with this book and the *BASIC Handbook*.

Related Publications

The following books contain related information that you may find useful:

- The IBM Personal Computer *Guide to Operations*.
- The IBM Personal Computer *Disk Operating System*.
- The IBM Personal Computer *Disk Operating System Technical Reference*.
- The IBM Personal Computer *Technical Reference*.

Summary of Changes

This summary first lists the changes that were made in BASIC release 2.Ø, and then lists the BASIC release 3.Ø changes.

Changes in BASIC 2.0 and BASIC 2.1

The changes that were made in BASIC release 2.Ø are briefly described in the material that follows.

- Three enhancements were made to the BASIC command line:

  - The optional parameter *max blocksize* was added to the **/M:** switch, allowing you to reserve space beyond BASIC's data segment for assembly language subroutines.

  - The ATN, COS, EXP, LOG, SIN, SQR, and TAN functions now allow you to calculate in double precision by specifying **/D** in the BASIC command line.

  - You can redirect standard input and standard output by specifying <*stdin* or>*stdout* in the BASIC command line.

- Pressing Ctrl-PrtSc causes text sent to your screen to also be sent to your system printer.

- The *filespec* syntax was expanded to allow the specification of a path for a device or file. All commands and statements that accept *filespec* also accept *path*. The commands that allow paths are

BLOAD, BSAVE, KILL, LOAD, MERGE, NAME, RUN, and SAVE. The statements that allow paths are CHAIN and OPEN.

- The DELETE command syntax was expanded to allow line deletions from the specified line to the end of the program.

- The **PE** option was added to the OPEN "COM... statement syntax to allow for parity checking.

- The PLAY statement has two new options. (For use in Advanced BASIC only.)

  – >*n* raises the octave and plays note *n*.

  – <*n* lowers the octave and plays note *n*.

- The DRAW statement has two new options. (For use in Advanced BASIC only.)

  – **TA**(*n*) turns angle *n* from -36Ø to 36Ø degrees.

  – **P** *paint,boundary* sets figure color and border color.

- The POINT function allows the form **v=POINT**(*n*), which returns the value of the current *x* or *y* graphics coordinate. (For use in Advanced BASIC only.)

- The RANDOMIZE statement allows double-precision expressions.

- The LINE statement has a new option, *style*, which uses hexadecimal values to plot a pattern of points on the screen. (For use in Advanced BASIC only.)

- The PAINT statement has a new feature, tiling, which allows you to paint an area with a pattern rather than just a solid color. (For use in Advanced BASIC only.)

- The ON KEY(n), KEY(n), and KEY statements now allow trapping of six additional definable keys, 15-2Ø. (For use in Advanced BASIC only.)

- The GET and PUT statements were enhanced to allow record numbers in the range 1 to 16,777,215 to accommodate large files with short record lengths.

- EOF(Ø) returns the end-of-file condition on standard input devices used with redirection of I/O.

- The LOF function returns the actual number of bytes allocated to a file.

- The graphics statements CIRCLE, DRAW, LINE, PAINT, POINT, PSET, PRESET, VIEW, and WINDOW now use line clipping instead of wraparound.

Three new functions were added:

- The PLAY(n) function returns the number of notes currently in the Music Background (MB) buffer. (For use in Advanced BASIC only.)

- The PMAP function maps an expression to world or physical coordinates. (For use in Advanced BASIC only.)

- The TIMER function returns the number of seconds that have elapsed since midnight or System Reset.

Four new statements were added:

- The ON PLAY statement allows continuous music to play while a program is running. (For use in Advanced BASIC only.)

- The ON TIMER statement transfers control to a given line number in a BASIC program when a defined period of time has elapsed. (For use in Advanced BASIC only.)

- The VIEW statement lets you define a viewport (or area) within the physical limits of the screen. (For use in Advanced BASIC only.)

- The WINDOW statement lets you redefine the coordinates of the screen or viewport. (For use in Advanced BASIC only.)

Three new commands were added:

- The CHDIR command allows you to change the current directory.

- The MKDIR command creates a directory on the specified disk.

- The RMDIR command removes a directory from the specified disk.

Changes in BASIC 3.0

The following changes have been made in BASIC release 3.Ø:

- Device support allows BASIC to communicate with user-installed device drivers through the IOCTL statement and IOCTL$ function.

- IOCTL and IOCTL$ are used to get information to and from device channels.

- ENVIRON statement and ENVIRON$ function allow you to modify parameters in BASIC's environment table.

- ERDEV and ERDEV$ are device error variables that allow you to read INT 24 error codes.

- SHELL allows you to execute DOS commands and run child processes from BASIC.

Note: The terms "disk," "diskette," and "fixed disk" are used throughout this book. Where "diskette" is used, it applies only to diskette drives and diskettes. Where "fixed disk" is used, it applies only to the IBM nonremovable fixed disk drive. Where "disk" is used, it applies to both fixed disks and diskettes.

Contents

BASIC Commands, Statements, and Functions

How to Use This Book

Descriptions of all the BASIC commands, statements, and functions are included in this book. The entries are arranged alphabetically. For more background and general information on BASIC, refer to the accompanying publication, the *BASIC Handbook*.

BASIC's built-in functions and variables can be used in any program without further definition.

The distinction between a command and a statement is largely a matter of tradition. Commands, because they generally operate on programs, are usually entered in direct mode. Statements generally direct program flow from within a program, and so are usually entered in indirect mode as part of a program line. Actually, most BASIC commands and statements can be entered in either direct or indirect mode.

The description of each command, statement, function, or variable in this section is formatted as follows:

Purpose: Tells what the command, statement, function, or variable does.

Versions: Indicates which versions of BASIC allow the command, statement, function, or variable. For example, if you look under "CHAIN Statement," you can see that after **Versions:** it says:

Cassette	Disk	Advanced	Compiler
	***	***	(**)

The asterisks indicate which versions of BASIC support the function. This example shows that you can use the CHAIN statement for programs written in the Disk BASIC and Advanced BASIC versions of BASIC.

In this example you will notice that the asterisks under the word "Compiler" are in parentheses. This means that that there are differences between the way the statement works under the BASIC interpreter and the way it works under the IBM Personal Computer BASIC Compiler. The IBM Personal Computer *BASIC Compiler* is an optional software package available from your IBM dealer. If you have the *BASIC Compiler*, the IBM Personal Computer *BASIC Compiler* manual explains these differences.

Format: Shows the correct format for the command, statement, function, or variable. A complete explanation of the syntax format is presented in the preface of this book. Remember to keep these rules rules in mind:

- Words in capital letters are keywords and must be entered as shown, except that they can be entered in any combination of uppercase and lowercase letters. BASIC automatically converts letters to uppercase (unless they are part of a quoted string, remark, or DATA statement).

- You are to supply any items shown in lowercase italic letters.

- Items in square brackets ([ ]) are optional.

- An ellipsis (...) indicates that an item can be repeated as many times as you wish.

- All punctuation except square brackets (such as commas, parentheses, semicolons, hyphens, or equal signs) must be included where shown.

Remarks: Describes in detail how to use the command, statement, function, or variable.

Example: Shows direct mode statements, sample programs, or program segments that demonstrate the use of the command, statement, function, or variable.

If a single- or double-precision value is supplied where an integer is required, BASIC rounds up the fractional portion and uses the resulting integer.

ABS
Function

Purpose: Returns the absolute value of the expression *x*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: $v = \text{ABS}(x)$

Remarks: *x* can be any numeric expression.

The absolute value of a number is always positive or zero.

Example: This example shows that the absolute value of -35 is positive 35.

```
PRINT ABS(7*(-5))
 35
```

ASC
Function

Purpose: Returns the ASCII code for the first character of a string (*x$*).

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = ASC(*x$*)

Remarks: *x$* can be any string expression.

The result of the ASC function is a numerical value that is the ASCII code of the first character of the string *x$*. See Appendix D for a list of ASCII codes. If *x$* is null, an **Illegal function call** error is returned.

The CHR$ function is the inverse of the ASC function, and it converts the ASCII code to a character.

Example: This example shows that the ASCII code for a capital **T** is 84. **PRINT ASC("TEST")** would work just as well.

```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
```

ATN
Function

Purpose: Returns the arctangent of *x*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = ATN(*x*)

Remarks: *x* can be a numeric expression of any type.

The ATN function returns the angle whose tangent is *x*. The result is a value in radians in the range -PI/2 to PI/2, where PI=3.141593.

If you want to convert radians to degrees, multiply by 18Ø/PI.

In BASIC 2.Ø and later releases, you can have this calculation performed in double-precision by specifying **/D** in the BASIC command line when BASIC is initially loaded. See "Options in the BASIC Command" in the *BASIC Handbook*.

Example: The first example shows the use of the ATN function to calculate the arctangent of 3.

```
PRINT ATN(3)
 1.249Ø46
```

ATN
Function

The second example finds the angle whose tangent is 1. It is .7853983 radians, or 45 degrees.

```
10 PI=3.141593
20 RADIANS=ATN(1)
30 DEGREES=RADIANS*180/PI
40 PRINT RADIANS,DEGREES
RUN
 .7853983      45
```

AUTO Command

Purpose: Automatically generates the next line number each time you press Enter.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: AUTO [*number*] [,[*increment*]]

Remarks:

number is the number used to start numbering lines. A period (.) can be used in place of the line number to indicate the current line.

increment is the value added to each line number to get the next line number.

Numbering begins at *number* and increases each subsequent line number by the value of *increment*. If both values are omitted, the default is 1Ø,1Ø. If *number* is followed by a comma but *increment* is not specified, the last increment specified in an AUTO command is assumed. If *number* is omitted but *increment* is included, then line numbering begins with Ø.

AUTO is used for entering programs. It saves having to type each line number.

If AUTO generates a line number that already exists in the program, an asterisk (*) is printed after the number to warn you that any input will replace the existing line. However, if you press Enter immediately after the asterisk, the existing line is not replaced, and AUTO generates the next line number.

AUTO ends when you press Ctrl-Break. The line in which Ctrl-Break is typed is not saved. After a Ctrl-Break, BASIC returns to command level.

> **Note:** When in AUTO mode, you can make changes only to the current line. If you want to change another line on the screen, be sure to exit AUTO by first pressing Ctrl-Break.

Example: This generates line numbers 1Ø, 2Ø, 3Ø, 4Ø, ...

```
AUTO
```

This generates line numbers 1ØØ, 15Ø, 2ØØ, ...

```
AUTO 1ØØ,5Ø
```

This generates line numbers Ø, 2Ø, 4Ø, 6Ø, ...

```
AUTO ,2Ø
```

If the increment in the previous AUTO command was 5Ø, then this command generates line numbers 5ØØ, 55Ø, 6ØØ, 65Ø, ...

```
AUTO 500,
```

BEEP
Statement

Purpose: Causes the speaker to sound.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: BEEP

Remarks: The BEEP statement causes the speaker to sound at 8ØØ Hz for 1/4 second. BEEP has the same effect as:

```
PRINT CHR$(7);
```

Example: In this example, the program checks to see if X is out of range. If it is, the computer warns you by beeping.

```
1ØØ IF X < 2Ø THEN BEEP
```

BLOAD Command

Purpose: Loads a memory image file into memory.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: BLOAD *filespec* [,*offset*]

Remarks:

filespec is a string expression for the file specification. In BASIC 2.Ø and later releases, it can contain a path. It must conform to the rules outlined under "Naming Files" in Chapter 3 of the *BASIC Handbook*; otherwise, an error occurs.

offset is an integer expression in the range Ø to 65535. This is an offset at which the file will be loaded into the current segment specified by the latest DEF SEG statement.

If *offset* is omitted, the offset specified at BSAVE is assumed. That is, the file is loaded into the same location from which it was BSAVED.

When a BLOAD command is executed, the named file is loaded into memory starting at the specified location. If the file is to be loaded from the device CAS1:, the cassette motor is turned on automatically.

If you are using Cassette Basic and the device name is omitted, CAS1: is assumed. CAS1: is the only device allowed for BLOAD in Cassette Basic. If you

are using Disk BASIC or Advanced BASIC and the device name is omitted, the DOS default drive is used.

BLOAD is intended for use with a file that has previously been saved with BSAVE. BLOAD and BSAVE are useful for loading and saving machine language programs, but they are not restricted to assembly language programs. Any segment can be specified as the target or source for these statements through the DEF SEG statement. You have a useful way of saving and displaying screen images: save from or load to the screen buffer. See also Appendix B, "Assembly Language Subroutines."

Warning:
BASIC does not check the offset of the current segment where you are BLOADing. That is, it is possible to use BLOAD anywhere in memory. Do not BLOAD over BASIC's stack, BASIC's variable area, or your BASIC program. See the memory map in Appendix B.

Notes when using CAS1::

1. If you enter the BLOAD command in direct mode, the file names on the tape are displayed on the screen followed by a period (.) and a single letter indicating the type of file. This is followed by the message **Skipped** for the files not matching the named file, and **Found** when the named file is found. Types of files and the associated letter are:

   .B for BASIC programs in internal format (created with SAVE command)

   .P for protected BASIC programs in internal format (created with SAVE ,P command)

.A for BASIC programs in ASCII format (created with SAVE ,A command)

.M for memory image files (created with BSAVE command)

.D for data files (created by OPEN followed by output statements)

If the BLOAD command is executed in a BASIC program, the file names skipped and found are not displayed on the screen.

2. You can press Ctrl-Break any time during BLOAD. This will cause BASIC to exit the search and return to direct mode between files or after a time-out period. Previous memory contents do not change.

3. If CAS1: is specified as the device and the filename is omitted, the next memory image (.M) file on the tape is loaded.

BLOAD Command

Example: This example loads the screen buffer for the Color/Graphics Monitor Adapter, which is at segment address HB8ØØØ. If you were loading the screen buffer for the IBM Monochrome and Parallel Printer Adapter, you would have to change line 3Ø to read &HBØØØ. Line 5Ø loads PICTURE at offset Ø, segment &HB8ØØ.

```
1Ø 'load the screen buffer
2Ø 'point SEG at screen buffer
3Ø DEF SEG= &HB8ØØ
4Ø 'load PICTURE into screen buffer
5Ø BLOAD "PICTURE",Ø
```

The example for the BSAVE command (see the next entry) illustrates how PICTURE was saved.

BSAVE Command

Purpose: Saves portions of the computer's memory on the specified device.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: BSAVE *filespec,offset,length*

Remarks:

filespec is a string expression for the file specification. In BASIC 2.Ø and later releases, it can contain a path. It must conform to the rules outlined under "Naming Files" in Chapter 3 of the *BASIC Handbook*; otherwise, an error occurs.

offset is an integer expression in the range Ø to 65535. This is the offset into the segment declared by the last DEF SEG. Saving starts from this location. See "DEF SEG Statement."

length is an integer expression in the range 1 to 65535. This is the length of the memory image to be saved.

If *offset* or *length* is omitted, a **Syntax error** occurs and the save is canceled.

In Cassette BASIC, if the device name is omitted, CAS1: is assumed. CAS1: is the only device allowed for BSAVE in Cassette Basic. In Disk BASIC and Advanced BASIC, if the device name is omitted, the DOS default disk drive is used.

BSAVE Command

If you are saving to CAS1:, the cassette motor is turned on and the memory image file is immediately written to the tape.

When you use the DEF SEG statement, you can specify any segment as the source segment for the BSAVE data. For example, you can save an image of the screen by doing a BSAVE of the screen buffer.

Example: As explained under "BLOAD Command", the segment address of the 16K screen buffer for Color/Graphic Monitor Adapter is HB8ØØØ. The segment address of the 4K screen buffer for the IBM Monochrome Display and Parallel Printer Adapter is HBØØØØ.

The DEF SEG statement must be used to set up the segment address to the start of the screen buffer. The offset of Ø and length &H4ØØØ specify that the entire 16K screen buffer is to be saved.

```
1Ø 'Save the color screen buffer
2Ø 'point segment at screen buffer
3Ø DEF SEG= &HB8ØØ
4Ø 'save buffer in file PICTURE
5Ø BSAVE "PICTURE",Ø,&H4ØØØ
```

CALL Statement

Purpose: Calls an assembly language subroutine.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: CALL *numvar* [(*variable* [,*variable*]....)]

Remarks:

numvar is the name of a numeric variable. The value of the variable indicates the offset of the subroutine into the current segment of memory as defined by the last DEF SEG statement.

variable is the name of a variable to be passed as an argument to the assembly language subroutine.

The CALL statement is a way of interfacing assembly language programs with BASIC. See Appendix B, "Assembly Language Subroutines," for specific considerations when using assembly language subroutines.

CALL Statement

Example: Line 1Ø sets the segment to BASIC's segment. Line 3Ø declares all scalar values and arrays used in the program. Line 4Ø computes the offset of ARRAY into BASIC'S data segment. Line 5Ø loads the file into an integer array, and line 6Ø calls the routine. The variables Q, B$, and C are passed as arguments to the routine.

```
1Ø DEF SEG: OPTION BASE 1
2Ø DEFINT A-Z
3Ø DIM ARRAY(512): P=Ø: Q=5: B$="TRUE":B$="TRUE": C=Ø
4Ø P=VARPTR(ARRAY(1))
5Ø BLOAD "ASM.FIL",P
6Ø CALL P(Q,B$,C)
```

CDBL Function

Purpose: Converts x to a double-precision number.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: v = CDBL(x)

Remarks: x can be any numeric expression.

Rules for converting from one numeric precision to another are followed as explained in Chapter 3 of the *BASIC Handbook*. See also "CINT" and "CSNG" functions for converting numbers to integer and single precision.

Example: The value of CDBL(A) is accurate only to the second decimal place after rounding. This is so because only two decimal places of accuracy are supplied with A.

```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
 454.67  454.669829100156
```

CHAIN Statement

Purpose: Transfers control to another program, and passes variables to it from the current program.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	(**)

Format: CHAIN [MERGE] *filespec* [,[*line*] [,[ALL] [,DELETE *range*]]]

Remarks: MERGE brings a section of code into the BASIC program as an overlay. That is, a MERGE operation is performed with the chaining program. The chained-to program must be an ASCII file if it is to be merged. Example:

```
CHAIN MERGE "A:OVRLAY",1ØØØ
```

filespec is a string expression for the file specification. In BASIC 2.Ø and later releases, it can contain a path. It must conform to the rules outlined under "Naming Files" in Chapter 3 of the *BASIC Handbook*; otherwise, an error occurs. The filename is the name of the program to which control is transferred. Example:

```
CHAIN "A:PROG1"
```

line is a line number or an expression that evaluates to a line number in the chained-to program. It specifies the line at which the chained-to program is to

begin running. If it is omitted, execution begins at the first line in the chained-to program.

line is not affected by a RENUM command. If PROG1 is renumbered, this example CHAIN statement should be changed to point to the new line number. Example:

```
CHAIN "A:PROG1",1000
```

ALL specifies that every variable in the current program is to be passed to the chained-to program. If the ALL option is omitted, you must include a COMMON statement in the chaining program to pass variables to the chained-to program. See "COMMON Statement." Example:

```
CHAIN "A:PROG1",1000,ALL
```

DELETE behaves like the DELETE command. As in the DELETE command, the line numbers specified as the first and last line of the range must exist, or an **Illegal function call** error occurs. After using an overlay, you will usually want to delete it so that a new overlay can be brought in. Example:

```
CHAIN MERGE "A:OVRLAY2",1000,DELETE 1000-5000
```

CHAIN
Statement

This example deletes lines 1ØØØ through 5ØØØ of the chaining program before loading in the overlay (chained-to program). The line numbers in *range* are affected by the RENUM command.

Notes:

1. The CHAIN statement leaves files open.

2. The CHAIN statement with MERGE option preserves the current OPTION BASE setting.

3. Without MERGE, CHAIN does not preserve variable types or user-defined functions for use by the chained-to program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEF FN statements containing shared variables must be restated in the chained program.

4. The CHAIN statement does a RESTORE before running the chained-to program.

CHDIR Command

Purpose: Changes the current directory. (For BASIC 2.Ø and later releases.)

Versions:

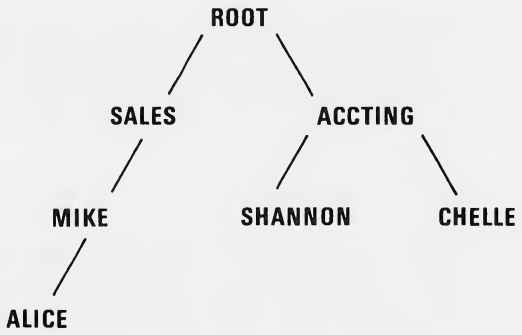
Cassette	Disk	Advanced	Compiler
	***	***	

Format: CHDIR *path*

Remarks:

path is a string expression, not exceeding 63 characters, identifying the new directory that will become the current directory. For more information on paths refer to "Naming Files" and "Tree-Structured Directories" in Chapter 3 of the *BASIC Handbook*.

Example:



CHDIR Command

(The examples that follow refer to the tree structure shown on the previous page.)

To change to the root directory from any subdirectory, use:

```
CHDIR "\"
```

To change to the directory ALICE from the root directory, use:

```
CHDIR "SALES\MIKE\ALICE"
```

To change to the directory CHELLE from the directory ACCTING, use:

```
CHDIR "CHELLE"
```

To change from the directory MIKE to the directory SALES, use:

```
CHDIR ".."
```

To make SALES the current directory on the current drive (drive A) and INVENTORY the current directory on drive C, use:

```
CHDIR "SALES"
CHDIR "C:INVENTORY"
```

The directory INVENTORY must exist on drive C. Now when you use *filespec* on drive A, it refers to the files in the directory SALES. When you use *filespec* on drive C, it refers to the files in the directory INVENTORY.

CHR$ Function

Purpose: Converts an ASCII code to its character equivalent.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = CHR$(*n*)

Remarks: *n* must be in the range Ø to 255.

The CHR$ function returns the one-character string with ASCII code *n*. ASCII codes are listed in Appendix D, "ASCII Character Codes." CHR$ is commonly used to send a special character to the screen or printer. For instance, the BEL character, which beeps the speaker, might be included as CHR$(7) as a preface to an error message (instead of using BEEP). See "ASC Function," earlier in this manual, for information on how to convert a character back to its ASCII code.

Example: This example prints the character equivalent of ASCII code 66.

```
PRINT CHR$(66)
B
```

The next example sets function key F1 to the string "AUTO" plus Enter. This is a good way to set the function keys so Enter is automatic when you press the function key.

```
KEY 1,"AUTO"+CHR$(13)
```

CHR$
Function

The following example is a program that shows all the displayable characters, along with their ASCII codes, on the screen in 8Ø-column width. It can be used with either the IBM Monochrome Display and Parallel Printer Adapter or the Color/Graphics Monitor Adapter.

```
1Ø CLS
2Ø FOR I=1 TO 255
3Ø ' ignore nondisplayable characters
4Ø IF (I>6 AND I<14) OR (I>27 AND I<32) THEN 1ØØ
5Ø COLOR Ø,7 ' black on white
6Ø PRINT USING "###"; I ; ' 3-digit ASCII code
7Ø COLOR 7,Ø ' white on black
8Ø PRINT " "; CHR$(I); " ";
9Ø IF POS(Ø)>75 THEN PRINT ' go to next line
1ØØ NEXT I
```

CINT Function

Purpose: Converts *x* to an integer.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = CINT(*x*)

Remarks:

x can be any numeric expression. If *x* is not in the range -32768 to 32767, an **Overflow** error occurs.

x is converted to an integer by rounding (up) the fractional portion.

See "FIX" and "INT" functions, both of which also return integers. See "CDBL" and "CSNG" functions for converting numbers to single or double precision.

Example: Observe in both examples how rounding occurs.

```
PRINT CINT(45.499)
 45

PRINT CINT(-2.89)
-3
```

CIRCLE Statement

Purpose: Draws an ellipse on the screen with center (*x,y*) and radius *r*.

Versions:

Cassette	Disk	Advanced	Compiler
		***	***

Graphics mode only.

Format: CIRCLE (*x,y*),*r* [,*color* [,*start,end* [,*aspect*]]]

Remarks:

(x,y) are the coordinates of the center of the ellipse. The coordinates can be given in either absolute or relative form. See "Specifying Coordinates" under "Graphics Modes" in Chapter 3 of the *BASIC Handbook*.

r is the radius (major axis) of the ellipse in points.

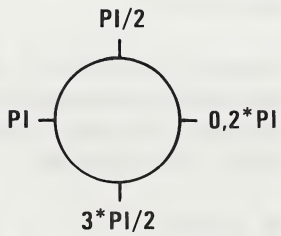
color is an integer expression that chooses a color attribute from the color attribute range for the current screen mode. In medium resolution, the color is the current one for that color attribute as defined by the COLOR statement. Four color attributes (Ø-3) are available in medium resolution; in high resolution, two attributes (Ø-1) are available. Zero (Ø) is always the color attribute for the background. The default foreground color attribute is always the maximum

color attribute for that screen mode: 3 in medium resolution; 1 in high resolution.

start, end are angles in radians and can range from -2*PI to 2*PI, where PI=3.141593.

aspect is a numeric expression.

start and *end* specify where the drawing of the ellipse will begin and end. The angles are positioned in the standard mathematical way, with Ø to the right and going counterclockwise:

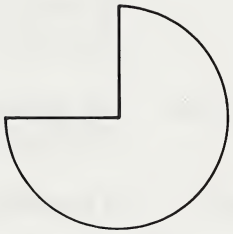


If the start or end angle is negative (-Ø is not allowed), the ellipse is connected to the center point with a line, and the angles are treated as if they were positive (note that this is not the same as adding 2*PI). The start angle can be greater or less than the end angle. For example,

```
1Ø PI=3.141593
2Ø SCREEN 1
3Ø CIRCLE (16Ø,1ØØ),6Ø,,-PI,-PI/2
```

draws a part of a circle similar to the following:

CIRCLE
Statement

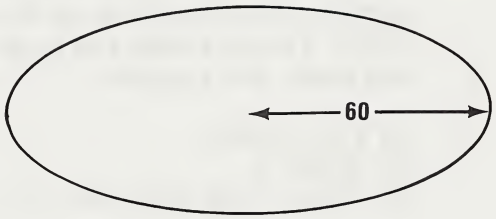


aspect affects the ratio of the x-radius to the y-radius. The default for *aspect* is 5/6 in medium resolution and 5/12 in high resolution. These values give a visual circle assuming the standard screen aspect ratio of 4/3.

If *aspect* is less than 1, then *r* is the *x*-radius. That is, the radius is measured in points in the horizontal direction. If *aspect* is greater than 1, then *r* is the *radius*. For example,

```
10 SCREEN 1
20 CIRCLE (160,100),60,,,,5/18
```

draws an ellipse like this:



In many cases, an *aspect* of 1 results in nicer-looking circles in medium resolution. It also causes the circle to be drawn somewhat faster.

The last point referenced after a circle is drawn is the center of the circle.

Points that are off the screen are clipped.

Example: The following example draws a face.

```
10 PI=3.141593
20 SCREEN 1 ' medium res. graphics
30 COLOR 0,1 ' black background, palette 1
40 'two circles in color 1 (cyan)
50 CIRCLE (120,50),10,1
60 CIRCLE (200,50),10,1
70 'two horizontal ellipses
80 CIRCLE (120,50),30,,,,5/18
90 CIRCLE (200,50),30,,,,5/18
100 'arc in color 2 (magenta)
110 CIRCLE (160,0),150,2, 1.3*PI, 1.7*PI
120 'arc, one side connected to center
130 CIRCLE (160,52),50,, 1.4*PI, -1.6*PI
```

CLEAR Command

Purpose: Sets all numeric variables to zero and all string variables to null. Options set the maximum number of bytes that BASIC will address and the amount of stack space.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: CLEAR [,[*n*] [,*m*]]

Remarks:

n is a byte count that, if specified, sets the maximum number of bytes for the BASIC data segment (where your program and data are stored along with the interpreter work area). The default value for *n* is 65535. You can specify *n* as a smaller value to decrease BASIC's total addressable space. This increases the amount of available memory beyond BASIC's data segment in high memory. Include *n* if you need to reserve space in storage for assembly language programs beyond the BASIC data segment.

m sets aside stack space for BASIC. The default is 512 bytes, or 1/8 of the available memory (whichever is smaller). Include *m* if you use many nested GOSUB statements or FOR...NEXT loops in your program, or if you use PAINT to do complex scenes.

CLEAR frees all memory used for data without erasing the program currently in memory. After a CLEAR, arrays are undefined; numeric variables have a value of zero; string variables have a null

value; and any information set with any DEF statement is lost. (This includes DEF FN, DEF SEG, and DEF USR, as well as DEFINT, DEFDBL, DEFSNG, and DEFSTR.)

Executing a CLEAR command turns off any sound that is running and resets to Music Foreground. Also, PEN and STRIG are reset to OFF.

The ERASE statement is useful to free some memory without erasing all the data in the program. It erases only specified arrays from the work area. See "ERASE Statement."

Example: This example clears all data from memory (without erasing the program):

```
CLEAR
```

The next example clears the data and sets the maximum data segment size to 32K bytes:

```
CLEAR ,32768
```

The next example clears the data and sets the size of the stack to 2ØØØ bytes:

```
CLEAR ,,2ØØØ
```

The last example clears data, sets the maximum data segment for BASIC to 32K bytes, and sets the stack size to 2ØØØ bytes:

```
CLEAR ,32768,2ØØØ
```

CLOSE
Statement

Purpose: Concludes I/O to a device or file.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: CLOSE [[#] *filenum* [,[#] *filenum*]....]

Remarks:

filenum is the number used on the OPEN statement.

The association between a particular file or device and its file number stops when CLOSE is executed. Subsequent I/O operations specifying that file number will be invalid. The file or device can be opened again using the same or a different file number; or the file number can be reused to open any device or file.

A CLOSE to a file or device opened for sequential output causes the final buffer to be written to the file or device.

A CLOSE with no file numbers specified causes all open devices and files to be closed.

Executing an END, NEW, RESET, SYSTEM, or RUN without the **R** option causes all open files and devices to be automatically closed. STOP does not close any files or devices.

See also "OPEN Statement" for information about opening files.

Example: This example causes the files and devices associated with file numbers 1, 2, and 3 to be closed.

```
100 CLOSE #1,#2,#3
```

This example causes all open devices and files to be closed.

```
100 CLOSE
```

CLS
Statement

Purpose: Clears the screen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: CLS

Remarks: If the screen is in text mode, the active page is cleared to the background color. See also "COLOR" and "SCREEN" statements.

If the screen is in graphics mode (medium or high resolution), the entire screen buffer is cleared to the background color.

The CLS statement also returns the cursor to the home position. In text mode, this means the cursor is located in the upper left-hand corner of the screen. In graphics mode, this means the "last point referenced" for future graphics statements is the point in the center of the screen: (16Ø,1ØØ) in medium resolution; (32Ø,1ØØ) in high resolution.

Changing the screen mode or width by using the SCREEN or WIDTH statements also clears the screen. The screen can also be cleared by pressing Ctrl-Home.

When you are using the VIEW statement, CLS clears only the last viewport. To clear the entire screen you must use VIEW to disable the active viewport, and then use CLS to clear the screen. (Viewports are used in BASIC 2.Ø and later releases.)

CLS Statement

Example: With the Color/Graphics Monitor Adapter, this example clears the screen to blue.

```
10 SCREEN 0,0,0
20 COLOR 10,1
30 CLS
```

COLOR
Statement

Purpose: Sets the colors for the foreground, background, and border screen. See "Text Mode" in Chapter 3 of the *BASIC Handbook* for an explanation of these terms.

The syntax of the COLOR statement depends on whether you are in text mode or graphics mode, as set by the SCREEN statement.

When BASIC is first started, the color is initially set to white on black.

In text mode, you can set the following:

Foreground-	1 of 16 color attributes Character blink, if desired
Background-	1 of 8 color attributes
Border-	1 of 16 color attributes

You can set the following in medium-resolution graphics mode:

Background-	1 of 16 color attributes
Palette-	1 of 2 palettes with 3 color attributes each

The border is the same as the background color.

COLOR Statement

The COLOR Statement in Text Mode

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: COLOR [*foreground*] [,[*background*] [,*border*]]

Remarks:

foreground is a numeric expression in the range Ø to 31, representing the character color.

background is a numeric expression in the range Ø to 7 for the background color.

border is a numeric expression in the range Ø to 15. It is the color for the border screen.

If you have the Color/Graphics Monitor Adapter, the following colors are allowed in *foreground*:

Ø Black	8 Gray
1 Blue	9 Light Blue
2 Green	1Ø Light Green
3 Cyan	11 Light Cyan
4 Red	12 Light Red
5 Magenta	13 Light Magenta
6 Brown	14 Yellow
7 White	15 High-intensity White

Colors and intensity can vary depending on your display device.

You might like to think of colors 8 to 15 as "light" or "high-intensity" values of colors Ø to 7.

COLOR Statement

You can make the characters blink by setting *foreground* equal to 16 plus the number of the desired color. That is, a value of 16 to 31 causes blinking characters.

You can select only colors Ø through 7 for *background* in text mode.

If you have the IBM Monochrome Display and Parallel Printer Adapter, the following values can be used for *foreground*:

Ø	Black
1	Underlined character with standard foreground color
2-7	Standard foreground color

With the Color/Graphics Monitor, adding 8 to the number of the desired color gives you the color in high intensity. For example, an attribute of 15 gives you the standard color in high intensity.

With the Color/Graphics Monitor Adapter, you can make the character blink by adding 16 to the attribute. Thus, 31 gives you high-intensity standard color characters.

For *background* with the IBM Monochrome Display and Parallel Printer Adapter, you can select the following values:

Ø–6	Black
7	Standard foreground color

Note: Attribute 7 as a background attribute appears as the standard color on the IBM Monochrome Display only when it is used with a foreground attribute of Ø, 8, 16, or 24 (black).

Conversely, black (attribute Ø, 8, 16, or 24) as a foreground attribute shows up as black only when used with a background attribute that creates reverse image characters. Black used with a background attribute of Ø makes the characters invisible.

Other combinations of foreground and background attributes produce standard results on the IBM Monochrome Display.

Notes for either adapter:

1. Foreground attribute can equal background attribute. This makes any character displayed invisible. Changing the foreground or background attribute makes subsequent characters visible again.

2. Any parameter can be omitted. Omitted parameters assume the old value.

3. If the COLOR statement ends in a comma (,), you get a **Missing operand** error, but the color changes. For example,

   ```
   COLOR 1,7,
   ```

   is invalid.

4. Any values entered outside the range Ø to 255 result in an **Illegal function call** error. Previous values are retained.

Example: This statement sets a yellow foreground, a blue background, and a black border screen.

```
1Ø COLOR 14,1,Ø
```

COLOR Statement

The following example can be used with either the Color/Graphics Monitor Adapter or the IBM Monochrome Display and Parallel Printer Adapter:

```
10 PRINT "Enter your ";
20 COLOR 15,0 'highlight next word
30 PRINT "password";
40 COLOR 7 'return to default (white on black)
50 PRINT " here: ";
60 COLOR 0 'invisible (black on black)
70 INPUT PASSWORD$
80 IF PASSWORD$="secret" THEN 120
90 ' blink and highlight error message
100 COLOR 31: PRINT "Wrong Password": COLOR 7
110 GOTO 10
120 COLOR 0,7 'reverse image (black on white)
130 PRINT "Program continues...";
140 COLOR 7,0 'return to default (white on black)
```

The COLOR Statement in Graphics Mode

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Graphics mode, medium resolution only.

Format: COLOR [*background*] [,[*palette*]]

Remarks:

background is an integer expression in the range Ø through 15. It specifies the background attribute.

palette is an integer expression. It selects one of two palettes of color.

In graphics mode, the COLOR statement sets a background color and chooses one of two palettes with four color attributes each (Ø-3). Color attribute Ø is always the current background. You can select one of three color attributes for the foreground color to be used with PSET, PRESET, LINE, CIRCLE, PAINT, VIEW, and DRAW. The COLOR statement in graphics mode has meaning only for medium resolution. Using COLOR in high resolution results in in an **Illegal function call** error. The colors selected when you choose each palette are as follows:

Color	**Palette Ø**	**Palette 1**
1	Green	Cyan
2	Red	Magenta
3	Brown	White

If *palette* is an even number, palette Ø is selected. This associates the colors green, red, and brown to the color attributes 1, 2, and 3.

If *palette* is an odd number, palette 1 (cyan/magenta/white) is selected.

Graphics mode can display text in any of the three colors available in the current palette. However, if you are not using a U.S. keyboard, refer to the "GRAFTABL Command" in *Disk Operating System Reference* for information regarding additional character support for the Color/Graphics monitor adapter and other keyboards.

You can change the foreground color of the characters from 3 to 2 to 1 by entering:

```
DEF SEG: POKE &HFE, COLOR
```

where COLOR is the attribute 1, 2, or 3; Ø is not allowed. Later PRINTs will use the specified color attribute.

The color selected for background can be the same as any of the palette colors.

Any parameter can be omitted from the COLOR statement. Omitting parameters will not cause the current background or palette to change.

Any values entered outside the range Ø to 255 cause an **Illegal function call** error. Previous values are retained.

Example: This statement sets the background to light blue and selects palette Ø.

```
1Ø SCREEN 1
2Ø COLOR 9,Ø
```

In the next example, the background stays light blue, and palette 1 is selected.

```
1Ø COLOR ,1
```

COM(n) Statement

Purpose: Enables or disables trapping of communications activity to the specified communications adapter.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: COM(*n*) ON

COM(*n*) OFF

COM(*n*) STOP

Remarks:

n is the number of the communications adapter (1 or 2).

A COM(*n*) ON statement must be executed to allow trapping by the ON COM(*n*) statement. If a nonzero line number is specified in the ON COM(*n*) statement, BASIC checks every time a new statement is executed to see if any characters have come in to the communications adapter.

If COM(*n*) is OFF, no trapping takes place, and any communication activity is not remembered even if it does take place.

If a COM(*n*) STOP statement has been executed, no trapping can take place. However, any communications activity that does take place is remembered so that an immediate trap occurs when COM(*n*) ON is executed.

COMMON Statement

Purpose: Passes variables to a chained program.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	(**)

Format: COMMON *variable*[,*variable*]...

Remarks:

variable is the name of a variable that to be passed to the chained-to program. Arrays are specified by appending "()" to the array name.

The COMMON statement is used with the CHAIN statement. COMMON statements can appear anywhere in a program, although it is recommended that they appear at the beginning. Any number of COMMON statements can appear in a program, but the same variable cannot appear in more than one COMMON statement. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Any arrays that are passed do not need to be dimensioned in the chained-to program.

Example: This example chains to program PROG3 on the disk in drive A, and passes the array D along with the variables A, BEE1, C, and G$.

```
100 COMMON A,BEE1,C,D(),G$
110 CHAIN "A:PROG3"
```

CONT
Command

Purpose: Resumes program execution after a break.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: CONT

Remarks: The CONT command can be used to resume program execution after Ctrl-Break has been pressed; a STOP or END statement has been executed; or an error has occurred. Execution continues at the point where the break occurred. If it occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt.

CONT is usually used with STOP for debugging. When execution is stopped, you can examine or change the values of variables by using direct mode statements. You can then use either CONT to resume execution, or a direct mode GOTO to resume execution at a particular line number.

CONT is invalid if the program has been edited during the break.

Example: The following example creates a long loop.

```
10 FOR A=1 TO 50
20 PRINT A;
30 NEXT A
RUN
 1  2  3  4  5  6  7  8  9  10 11 12
 13  14  15  16  17  18  19  20 21 22
 23  24  25  26  27  28  29
```

(At this point we interrupt the loop by pressing Ctrl-Break.)

```
   .
   .
   .
Break in 20
CONT
 30  31  32  33  34  35  36  37  38  39
 40  41  42  43  44  45  46  47  48  49
 50
```

COS
Function

Purpose: Returns the trigonometric cosine function.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: $v = \mathrm{COS}(x)$

Remarks:

x is the angle whose cosine is to be calculated. The value of x must be in radians. To convert from degrees to radians, multiply the degrees by PI/18Ø, where PI=3.141593.

In BASIC 2.Ø and later releases, you can have this calculation performed in double-precision by specifying **/D** in the BASIC command line when BASIC is initially loaded. See "Options in the BASIC Command" in *BASIC Handbook*.

Example: This example shows that the cosine of PI radians is equal to -1. Then it calculates the cosine of 18Ø degrees by first converting the degrees to radians (18Ø degrees happens to be the same as PI radians).

```
1Ø PI=3.141593
2Ø PRINT COS(PI)
3Ø DEGREES=18Ø
4Ø RADIANS=DEGREES*PI/18Ø
5Ø PRINT COS(RADIANS)
RUN
-1
-1
```

CSNG Function

Purpose: Converts *x* to a single-precision number.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = CSNG(*x*)

Remarks:

x is a numeric expression that will be converted to single precision.

The rules outlined under "How BASIC Converts Numbers from One Precision to Another" in Chapter 3 of the *BASIC Handbook* are used for the conversion.

See also "CINT" and "CDBL" functions for information on converting numbers to the integer and double-precision data types.

Example: In this example the value of the double-precision number A# is rounded at the 7th digit and returned as CSNG(A#).

```
10 A# = 975.3421222#
20 PRINT A#; CSNG(A#)
RUN
 975.3421222  975.3421
```

CSRLIN
Variable

Purpose: Returns the vertical coordinate of the cursor.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = CSRLIN

Remarks: The CSRLIN variable returns the current line (row) position of the cursor on the active page. The active page is explained under "SCREEN Statement." The value returned is in the range 1 to 25.

The POS function returns the column location of the cursor. See "POS Function."

See also "LOCATE Statement" to see how to set the cursor line.

Example: This example saves the cursor coordinates in the variables X and Y, then moves the cursor to line 24 to put the words "HI MOM" on that line. Then the cursor is moved back to its former position.

```
10 Y = CSRLIN 'record current line
20 X = POS(0) 'record current column
30 LOCATE 24,1: PRINT "HI MOM"
40 LOCATE Y,X 'restore position
```

CVI, CVS, CVD Functions

Purpose: Converts string variable types to numeric variable types.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format:

v = CVI(*2-byte string*)

v = CVS(*4-byte string*)

v = CVD(*8-byte string*)

Remarks: Numeric values read from a random file must be converted from strings into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single-precision number. CVD converts an 8-byte string to a double-precision number.

The CVI, CVS, and CVD functions do *not* change the bytes of the actual data. They change only the way BASIC interprets those bytes.

See also "MKI$, MKS$, MKD$ Functions," as well as Appendix A, "BASIC Disk Input and Output," in the *BASIC Handbook*.

CVI, CVS, CVD Functions

Example: This example uses a random file (#1), which has fields defined as in line 1ØØ. Line 11Ø reads a record from the file. Line 12Ø uses the CVS function to interpret the first 4 bytes (N$) of the record as a single-precision number. N$ was probably originally a number that was written to the file using the MKS$ function.

```
1ØØ FIELD #1,4 AS N$, 12 AS B$
11Ø GET #1
12Ø Y=CVS(N$)
```

DATA Statement

Purpose: Stores the numeric and string constants that are accessed by a program's READ statements.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: DATA *constant*[,*constant*]...

Remarks:

constant can be a numeric or string constant. No expressions are allowed in the list. The numeric constants can be in any format – integer, fixed point, floating point, hex, or octal. String constants in DATA statements do not have to be enclosed by quotation marks, unless the string contains commas, colons, or significant leading or trailing blanks.

DATA statements are nonexecutable and can be placed anywhere in the program. A DATA statement can contain as many constants as will fit on a line, and any number of DATA statements can be used in a program. The information contained in the DATA statements can be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program. The READ statements access the DATA statements in line-number order.

The variable type (numeric or string) in the READ statement must agree with the corresponding constant in the DATA statement or a **Syntax error** occurs.

DATA Statement

You cannot use the single quote (') to add comments to the end of a DATA statement. If you do, BASIC thinks it is part of a string. You can, however, use :REM to add a remark.

Use the RESTORE statement to reread information from any line in the list of DATA statements. See "RESTORE Statement."

Example: See examples under "READ Statement."

DATE$ Variable and Statement

Purpose: Sets or retrieves the date.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: As a variable:

*v$* = DATE$

As a statement:

DATE$ = *x$*

Remarks: **For the variable (v$ = DATE$):**

A 10-character string in the form *mm-dd-yyyy* is returned. Here, *mm* represents 2 digits for the month, *dd* is the day of the month (also 2 digits), and *yyyy* is the year. The date can have been set by DOS before entering BASIC.

For the statement (DATE$ = x$):

x$* is a string expression used to set the current date. You can enter *x$ in any one of the following forms:

mm-dd-yy
mm/dd/yy
mm-dd-yyyy
mm/dd/yyyy

The year must be in the range 198Ø to 2Ø99. If you use only one digit for the month or day, a Ø (zero) is assumed in front of it. If you enter only 1 digit for

the year, a zero is appended to make it 2 digits. If you enter only 2 digits for the year, the year is assumed to be 19*yy*.

Example: In this example we set the date to August 29, 1984. Notice how, when we read the date back using the DATE$ function, a zero is included in front of the month to make it 2 digits, and the year becomes 1984. Also, the month, day, and year are separated by hyphens even though we enter them as slashes.

```
10 DATE$=  "8/29/84"
20 PRINT DATE$
RUN 08-29-1984
```

DEF FN Statement

Purpose: Defines and names a function that you write.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: DEF FN*name*[(*arg* [,*arg*]...)] =*expression*

Remarks:

name is a valid variable name. This name, preceded by FN, becomes the name of the function.

arg is an argument. It is a variable name in the function definition that is replaced with a value when the function is called. The arguments in the list represent, on a one-to-one basis, the values that are given when the function is called.

expression defines the returned value of the function. The type of the *expression* (numeric or string) must match the type declared by *name*.

The definition of the function is limited to one statement. Arguments (*arg*) that appear in the function definition serve only to define the function; they do not affect program variables that have the same name. A variable name used in the *expression* does not have to appear in the list of arguments. If it does, the value of the argument is supplied when the function is called. Otherwise, the current value of the variable is used.

DEF FN Statement

The function type determines whether the function returns a numeric or string value. The type of function is declared by *name*, in the same way as variables are declared. See "How to Declare Variable Types" in Chapter 3 of the *BASIC Handbook*. If the type of *expression* (string or numeric) does not match the function type, a **Type mismatch** error occurs. If the function is numeric, the value of the expression is converted to the precision specified by *name* before it is returned to the calling statement.

A DEF FN statement must be executed to define a function before you can call that function. If a function is called before it has been defined, an **Undefined user function** error occurs. On the other hand, a function can be defined more than once. The most recently executed definition is used.

Recursive functions are not supported.

DEF FN is invalid in direct mode.

Example: In this example, line 2Ø defines the function FNAREA, which calculates the area of a circle with radius R. The function is called in line 4Ø.

```
1Ø PI=3.141593
2Ø DEF FNAREA(R)=PI*R^2
3Ø INPUT  "Radius? ",RADIUS
4Ø PRINT  "Area is " FNAREA(RADIUS)
RUN
Radius?
```

(Suppose you respond with 2)

```
Radius? 2
Area is 12.56637
```

Here is an example with two arguments:

```
10 DEF FNMUD(X,Y)=X-(INT(X/Y)*Y)
20 A = FNMUD(7.4,4)
30 PRINT A
RUN
 3.4
```

DEF SEG Statement

Purpose: Defines the current segment of memory. A subsequent BLOAD, BSAVE, CALL, PEEK, POKE, or DEF USR definition specifies the offset into this segment.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: DEF SEG [=*segment*]

Remarks:

segment is a numeric expression in the range Ø to 65535.

The initial setting for the segment when BASIC is started is BASIC's data segment (DS). BASIC's data segment is the beginning of your user workspace in memory. If you execute a DEF SEG statement that changes the segment, the value is *not* reset to BASIC's DS when you issue a RUN command.

If *segment* is omitted from the DEF SEG statement, the segment is set to BASIC's data segment. The value of BASIC's data segment can be found in segment Ø, offsets &H51Ø and &H511.

If *segment* is given, it should be a value based upon a 16-byte boundary, since segments begin only on paragraph boundaries. The value is shifted left 4 bits (multiplied by 16) to form the segment address for the subsequent operation. That is, if *segment* is in hexadecimal, a Ø (zero) is added to get the actual segment address. BASIC does not perform any checking to ensure that the segment value is valid.

DEF and SEG must be separated by a space; otherwise, BASIC interprets the statement **DEFSEG=1ØØ** to mean "Assign the value 1ØØ to the variable DEFSEG."

Any value entered outside the range indicated results in an **Illegal function call** error. The previous value is retained.

See also Appendix B, "Assembly Language Subroutines," for more information on using DEF SEG.

Example: The first example restores a segment to BASIC's data segment.

```
DEF SEG ' restore segment to BASIC's data segmer
```

In the second example, the screen buffer for the Color/Graphics Monitor adapter is at segment B8ØØ hex, offset Ø. Since segments are specified on 16-byte boundaries, the last hex digit is dropped on the DEF SEG specification.

```
DEF SEG=&HB8ØØ
```

DEFtype Statements

Purpose: Declares variable types as integer, single-precision, double-precision, or string.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: DEF*type* *letter*[-*letter*] [,*letter* [-*letter*]]...

Remarks:

type is INT, SNG, DBL, or STR.

letter is a letter of the alphabet (A-Z).

A DEF*type* statement declares that the variable names beginning with the letter or letters specified will be that type of variable. However, a type-declaration character (%, !, #, or $) always takes precedence over a DEF*type* statement in the typing of a variable. See "How to Declare Variable Types" in Chapter 3 of the *BASIC Handbook*.

If no type-declaration statements are encountered, BASIC assumes that all variables without declaration characters are single-precision variables.

If you use type-declaration statements, put them at the beginning of the program. The DEF*type* statement must be executed before you use any variables it declares.

Example: In this example, line 1Ø declares that all variables beginning with the letter L, M, N, O, or P are double-precision variables.

Line 2Ø causes all variables beginning with the letter A to be string variables.

Line 3Ø declares that all variables beginning with the letter X, D, E, F, G, or H are integer variables.

```
1Ø DEFDBL L-P
2Ø DEFSTR A
3Ø DEFINT X,D-H
4Ø ORDER = 1#/3: PRINT ORDER
5Ø ANIMAL =  "CAT": PRINT ANIMAL
6Ø X=1Ø/3: PRINT X
RUN
 .3333333333333333
CAT
 3
```

DEF USR Statement

Purpose: Specifies the location in memory of an assembly language subroutine, which is later called by the USR function.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: DEF USR[*n*]=*offset*

Remarks:

n must be a digit from Ø to 9. It identifies the number of the USR routine whose location in memory is being specified. If *n* is omitted, DEF USRØ is assumed.

offset is an integer expression in the range Ø to 65535. The value of *offset* is added to the current segment value to obtain the actual starting address of the USR routine. See "DEF SEG Statement."

Any number of DEF USR statements can appear in a program, thus allowing access to as many subroutines as necessary. The most recently executed value is used for the offset.

Example: This example loads an assembly language subroutine into an integer array. The *n* in line 6Ø is determined by the size of the subroutine. The offset passed to DEF USR is the offset into BASIC's data segment.

```
1Ø OPTION BASE 1
2Ø DEFINT A-Z
3Ø 'Define all variables before VARPTR
4Ø SUBRT=Ø: I=Ø: J=Ø
5Ø 'Dimension array for subroutine
6Ø DIM ARRAY (n)
7Ø 'Obtain offset of 1st array element into
   BASIC's data segment
8Ø SUBRT = VARPTR(ARRAY(1))
9Ø 'Load routine into the integer array
1ØØ BLOAD "ASMFILE",SUBRT
   .
   .
   .
1ØØØ 'Pass offset to DEF USR
1Ø1Ø DEF USRØ = VARPTR(ARRAY(1))
1Ø2Ø 'Execute the subroutine
1Ø3Ø J = USRØ(I)
```

DELETE Command

Purpose: Deletes program lines.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: DELETE [*line1*] [-*line2*]

DELETE [*line1*-]

Remarks:

line1 is the line number of the first line to be deleted.

line2 is the line number of the last line to be deleted.

The DELETE command erases the specified range of lines from the program. BASIC always returns to command level after a DELETE is executed.

DELETE *line1*- deletes all lines from the specified line number through the end of the program. This form is valid for BASIC 2.Ø and later releases.

A period (.) can be used in place of the line number to indicate the current line. If you specify a line number that does not exist in the program, an **Illegal function call** error occurs.

Example: This example deletes line 4Ø:

```
DELETE 4Ø
```

This example deletes lines 4Ø through 1ØØ, inclusive:

```
DELETE 4Ø-1ØØ
```

This example deletes line 4Ø through the end of the program:

```
DELETE 4Ø-
```

The last example deletes all lines up to and including line 4Ø:

```
DELETE -4Ø
```

DIM Statement

Purpose: Specifies the maximum values for array variable subscripts and allocates storage accordingly.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: DIM *variable*(*subscripts*)[,*variable*(*subscripts*)]...

Remarks:

variable is the name used for the array.

subscripts is a list of numeric expressions, separated by commas, which define the dimensions of the array.

When executed, the DIM statement sets all the elements of the specified numeric arrays to an initial value of zero. String array elements are all variable length, with an initial null value (zero length).

If an array variable name is used without a DIM statement, the maximum value of its subscript is assumed to be 1Ø. If a subscript is greater than the maximum specified, a **Subscript out of range** error occurs.

The minimum value for a subscript is always Ø, unless otherwise specified with the OPTION BASE statement. (See "OPTION BASE Statement.") The maximum number of dimensions for an array is 255.

If you try to dimension an array more than once, a **Duplicate definition** error occurs. You can, however, use the ERASE statement to erase an array so you

can dimension it again. For more information about arrays, see "Arrays" in Chapter 3 of the *BASIC Handbook*.

Example: This example creates two arrays: a one-dimensional numeric array named SIS with 13 elements, SIS(Ø) through SIS(12); and a two-dimensional string array named WRR$, with three rows and three columns.

```
1Ø WRRMAX=2
2Ø DIM SIS(12), WRR$(WRRMAX,2)
3Ø DATA 26.5, 37, 8, 29, 8Ø, 9.9, &H8ØØ
4Ø DATA 7, 18, 55, 12, 5, 43
5Ø FOR I=Ø TO 12
6Ø READ SIS(I)
7Ø NEXT I
8Ø DATA SHERRY, ROBERT,  "A:"
9Ø DATA  "HI, LYNN", HELLO, GOOD-BYE
1ØØ DATA BOCA RATON, DELRAY, MIAMI
11Ø FOR I=Ø TO 2: FOR J=Ø TO 2
12Ø READ WRR$(I,J)
13Ø NEXT J,I
14Ø PRINT SIS(3); WRR$(2,Ø)
RUN
 29 BOCA RATON
```

DRAW Statement

Purpose: Draws an object as specified by *string*.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Graphics mode only.

Format: DRAW *string*

Remarks: The DRAW statement draws objects using a *graphics definition language*. The language commands are contained in the string expression *string*. The string defines an object, which is drawn when BASIC executes the DRAW statement. During execution, BASIC examines the value of *string* and interprets single-letter commands from the contents of the string. When a movement command is given, a line is drawn from the last point referenced.

n in the following movement commands indicates the distance to move. The number of points moved is *n* times the scaling factor (set by the **S** command). The movement commands are detailed on the next page.

U n	Move up.
D n	Move down.
L n	Move left.
R n	Move right.
E n	Move diagonally up and right.
F n	Move diagonally down and right.
G n	Move diagonally down and left.
H n	Move diagonally up and left.
M x,y	Move absolute or relative. If *x* has a plus sign (+) or a minus sign (-) in front of it, it is relative. Otherwise, it is absolute.

The following two prefix commands can precede any of these movement commands:

B	Move, but don't plot any points.
N	Move, but return to the original position when finished.

The following commands are also available:

A n	Set angle *n*. The value of *n* can range from Ø to 3, where Ø is Ø degrees, 1 is 9Ø, 2 is 18Ø, and 3 is 27Ø. Figures rotated 9Ø or 27Ø degrees are scaled so they appear the same size with Ø or 18Ø degrees on a display screen with standard aspect ratio 4/3.

DRAW Statement

TA n Turn angle *n*. The value of *n* can range from -36Ø to +36Ø. If *n* is positive (+), the angle turns counterclockwise. If *n* is negative (-), the angle turns clockwise. Values entered that are outside of the range -36Ø to +36Ø cause an **Illegal function call** error. This command is valid for BASIC version 2.Ø and later releases.

C n Set color *n*. The value of *n* can range from Ø to 3 in medium resolution, and Ø to 1 in high resolution. In medium resolution, *n* selects the color attribute from the current palette as defined by the COLOR statement. Zero (Ø) is always the attribute for the background. The default foreground color attribute is always the maximum attribute for that screen mode: 3 in medium resolution; 1 in high resolution.

S n Set scale factor. The value of *n* can range from 1 to 255. The scale factor is *n* divided by 4. For example, if *n*=1, then the scale factor is 1/4. The scale factor multiplied by the distances given with the U, D, L, R, E, F, G, H, and relative M commands gives the actual distance moved. The default value is 4, so the scale factor is 1.

X variable;
Execute substring. This allows you to execute a second string from within a string.

P paint,boundary

Set figure color to *paint* and border color to *boundary*. The *paint* parameter is an integer expression. It chooses an attribute from the attribute range for the current screen mode. In medium resolution, this color is one from the current palette as defined for that attribute by the COLOR statement. Four color attributes (Ø-3) are available in medium resolution. In high resolution, two color attributes (Ø-1) are available: Ø indicates black and 1 indicates white. The *boundary* parameter is the border color of the figure to be filled in, in the attribute range for the current screen mode. You must specify both *paint* and *boundary*, or an error occurs. This command does not support paint tiling, and it is valid for BASIC 2.Ø and later releases.

In all these commands, the *n, x*, or *y* argument can be a constant such as **123** or it can be **=*variable*;** where *variable* is the name of a numeric variable. The semicolon (;) is required when you use a variable this way, or in the **X** command. Otherwise, a semicolon is optional between commands. Spaces are ignored in *string*. For example, you can use variables in a move command this way:
M+=X1; -=X2;

You can also specify variables in the form VARPTR$(*variable*), instead of **=*variable*;**. This is the only form that can be used in compiled programs. For example:

DRAW Statement

One Method	Alternative Method
DRAW "XA$;"	DRAW "X"+VARPTR$(A$)
DRAW "S=SC;"	DRAW "S="+VARPTR$(SC)

The **X** command can be a very useful part of DRAW. It allows you to define segments of a picture in different **X** variables and to combine these **X** variables into a single DRAW statement. In this way you are able to create DRAW strings longer than 255 characters. For example, if you are creating a scene of a house with a chimney and a tree, each of these objects can be defined in an **X** variable so your DRAW statement can look like this:

```
DRAW "XHOUSE$;XCHIM$;XTREE$;"
```

The aspect ratio of your screen determines the spacing of the horizontal, vertical, and diagonal points. The DRAW statement does not take into account the aspect ratio of the current screen mode; that is, DRAW "R5Ø U5Ø" plots exactly 5Ø points to the right and then 5Ø up, but the two lines will not appear to be equal in length.

The aspect ratio is used to correct the shape of objects drawn on a nonlinear surface. The idea is to be able to draw a square, for example, that indeed looks square.

DRAW Statement

If there are 64Ø by 64Ø dots on a screen evenly spaced along the *x* and *y* axes, the aspect ratio is "1 to 1" or 1/1. This is an ideal surface. If you execute the statement:

```
DRAW  "R1ØØ D1ØØ L1ØØ U1ØØ"
```

then the box appears square.

However, this is not the case in BASIC, which supports two screen resolutions, each with its own aspect ratio. These are:

Resolution	Aspect Ratio	
Medium resolution	32Ø by 2ØØ dots	5/6
High resolution	64Ø by 2ØØ dots	5/12

To draw a box that appears square in either resolution, scale the *y* axis by the corresponding aspect ratio; or scale the *x* axis by 1/aspect ratio.

For example, to draw a square box 1ØØ high, scale the *x* axis as follows:

```
1Ø '1ØØ*6/5 is 12Ø
2Ø DRAW "U1ØØ R12Ø D1ØØ L12Ø"
```

DRAW Statement

Example: To draw a box using variables:

```
10 SCREEN 1
20 A=2030 DRAW "U=A;R=A;D=A;L=A;"
```

To draw a box and paint the interior:

```
10 DRAW "U50R50D50L50" 'Draw a box
20 DRAW "BE10" 'Move up and right into box
30 DRAW "P1,3" 'Paint interior
```

To draw a triangle:

```
10 SCREEN 1
20 DRAW "E15 F15 L30"
```

To create a "shooting star":

```
10 SCREEN 1,0: COLOR 0,0: CLS
20 DRAW "BM300,25" ' initial point
30 STAR$= "M+7,17 M-17,-12 M+20,0 M-17,12 M+7,-17"
40 FOR SCALE=1 TO 40 STEP 2
50 DRAW "C1;S=SCALE; BM-2,0;XSTAR$;"
60 NEXT
```

To draw some spokes:

```
10 SCREEN 1,0:CLS
20 FOR D=0 TO 360 STEP 10
30 DRAW "TA=D; NU50"
40 NEXT D
```

EDIT Command

Purpose: Displays a line for editing.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: EDIT *line*

Remarks:

line is the line number of a line existing in the program. If there is no such line, an **Undefined line number** error occurs.

EDIT simply displays the line specified and positions the cursor under the first digit of the line number. The line can then be modified as described under "The BASIC Program Editor" in the *BASIC Handbook*.

A period (.) can be used for the line number to refer to the current line. For example, if you have just entered a line and wish to go back and change it, the command **EDIT** redisplays the line for editing.

LIST can also be used to display program lines for changing. See "LIST Command."

END
Statement

Purpose: Terminates program execution, closes all files, and returns to command level.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: END

Remarks: END statements can be placed anywhere in the program to terminate execution. END is different from STOP in two ways:

- END does not cause a **Break** message to be printed.
- END closes all files.

An END statement at the end of a program is optional. BASIC always returns to command level after an END is executed.

Example: This example ends the program if K is greater than 1ØØØ; otherwise, the program branches to line number 2Ø.

```
1ØØ IF K>1ØØØ THEN END ELSE GOTO 2Ø
```

ENVIRON Statement

Purpose: Modifies parameters in BASIC's environment table. ENVIRON is used to change the "PATH" parameter for a child process or to pass parameters to a child process by inventing a new environment parameter. See ENVIRON$, SHELL, and the DOS PATH Command.

Not valid for releases earlier than 3.Ø.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	

Format: ENVIRON *parm* = *string*

Remarks:

parm is the name of the parameter, such as "PATH".

string is the text that defines the new parameter.

parm must be separated from *string* by an equal sign or a blank. ENVIRON takes everything left of the first blank or equal sign as *parm*. The first "nonblank, nonequal" after *parm* is taken as *string*.

If *string* is a null string or consists only of ";" (a single semicolon), such as:

```
"PATH=;"
```

the parameter is removed from the environment table and the table is compressed.

If *parm* does not exist, the new parameter is added at the end of the environment table.

If *parm* exists, it is deleted, the environment table is compressed, and *parm* is added at the end.

> **Note:** When BASIC is invoked, the size of its environment table is the current size of DOS's environment table (rounded up to the next 16-byte paragraph boundary.) BASIC cannot expand its environment table. If you wish to add elements to BASIC's environment table, you must expand the table from DOS to the size your application needs before invoking BASIC.

Example: You can create a default PATH to the root directory on drive A with the following statement:

```
ENVIRON "PATH=A:\"
```

Now, you can invoke DOS from your BASIC program using the SHELL statement and issue any valid DOS command. If a disk file is needed to execute the command, DOS now automatically searches for it (.COM, .EXE, or .BAT) in the root directory on drive A if it is not on the current drive or directory.

```
SHELL           'Invokes a copy of COMMAND.COM.

A> REM Changes directory to "WORK" on drive B.
A> CD B:\WORK
B> REM Loads PROJ1 under DEBUG even though no
   drive is specified.  DEBUG and PROJ1 are
   located on different drives.
B> DEBUG PROJ1
      .
      .
      .
B> REM Return to BASIC program
B> EXIT (exit DOS, return to BASIC program.)
```

You can add a new parameter to the environment table:

```
ENVIRON "HELP = C:\HELP" 'defines
                 file parameter called "HELP"
CHDIR ENVIRON$ ("HELP") 'changes dir to "HELP"
```

You can delete this parameter in the table by:

```
ENVIRON "HELP=;" 'deletes parameter "HELP"
                      from table
```

The environment you create from your BASIC application is passed to COMMAND.COM when it is invoked by the SHELL statement. This makes it possible to pass parameters from a parent (BASIC) to a child through the environment table.

> **Note:** For related information, see also "ENVIRON$ Function" and "SHELL Statement" in this manual. Also "SET Command" in *Disk Operating System Reference* and the "EXEC Function Call" in *Disk Operating System Technical Reference*

ENVIRON$
Function

Purpose: Retrieves and displays the specified string from BASIC's environment table.

Not valid for BASIC releases earlier than 3.Ø.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	

Format:

*v$* = ENVIRON$ (*parm*)
or
*v$* = ENVIRON$ (*n*)

Remarks:

parm	is a string expression containing the parameter to be retrieved.
n	is an integer expression returning a value in the range 1 to 255.

If a string argument is used, ENVIRON$ returns, from the environment table, a string containing the text that follows *parm*. If *parm* is not found or no text follows the equal sign, the null string is returned.

If a numeric argument is used, ENVIRON$ returns a string containing the *nth parm* from the environment table, along with the *parm*= text. If there is no *nth parm*, a null string is returned.

ENVIRON$ distinguishes between uppercase letters and lowercase letters. If you add to the table in this format:

```
ENVIRON$ "load = high"
```

and want to check to see if the operation was successful, you can use the ENVIRON$ function like this:

```
PRINT ENVIRON ("load")
```

But if you type:

```
PRINT ENVIRON$ ("LOAD")
```

ENVIRON$ returns a null string because 'LOAD" is not in the table; however, "load" *is* in the table.

Example: When DOS loads initially, it sets a parameter called "COMSPEC" that tells DOS where to locate the COMMAND.COM file, and it sets up a null path. To observe the contents of the environment table at start-up time, enter the following from BASIC:

```
PRINT ENVIRON$ (1)
```

You now see printed on the screen:

```
PATH=
```

If you enter:

```
PRINT ENVIRON$ (2)
```

you see displayed:

```
COMSPEC = A:\COMMAND.COM
```

Note: If you booted from a fixed disk, the previous example displays *C:* instead of *A:* for the drive specification.

ENVIRON$
Function

If you enter:

```
PRINT ENVIRON$ ("COMSPEC")
```

the computer's response is:

```
A:\COMMAND.COM
```

The following program saves BASIC's environment table in an array so that it can be modified for a child process. After the child process is completed, the environment is restored.

```
10 DIM TABLE$(10) 'assume no more than 10 parms
20 PARMS = 1     'initial number of parameters
30 WHILE LEN(ENVIRON$(PARMS)) > 0
40 TABLE$(PARMS) = ENVIRON$(PARMS)
50 PARMS = PARMS+1
60 WEND
70 PARMS = PARMS - 1  'adjust to correct number
80 'now store new environment
90 ENVIRON "DATAIN = C:\DATAIN\INP.FIL"
100 ENVIRON "SORT.DAT = SORT.DAT<" +
   ENVIRON$ ("DATAIN") +">LPT1:"
   .
   .
   .
1000 SHELL ENVIRON$("SORT.DAT") 'data is sorted
1010 FOR I = 1 TO PARMS
1020 ENVIRON TABLE$(I)  'restore parameters
1030 NEXT I
   .
   .
   .
```

Note: See also "ENVIRON Statement" and "SHELL Command." Also "SET Command" in *Disk Operating System Reference* and "EXEC Function Call" in *Disk Operating System Technical Reference*.

EOF
Function

Purpose: Indicates an end-of-file condition.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: *v* = EOF(*filenum*)

Remarks:

filenum is the number specified on the OPEN statement.

The EOF function is useful for avoiding an **Input past end** error. EOF returns -1 (true) if end of file has been reached on the specified file. A Ø (zero) is returned if end of file has not been reached.

EOF is significant only for a file opened for sequential input from disk or cassette, or for a communications file. A -1 for a communications file means the buffer is empty.

EOF
Function

Example: This example reads information from the sequential file named "DATA". Values are read into the array M until end of file is reached.

```
10 OPEN "DATA" FOR INPUT AS #1
20 C=0
30 IF EOF(1) THEN END
40 INPUT #1,M(C)
50 C=C+1: GOTO 30
```

EOF(Ø) returns the end-of-file condition on standard input devices used with redirection of I/O. (For BASIC 2.Ø and later releases.)

ERASE Statement

Purpose: Eliminates arrays from a program.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: ERASE *arrayname*[,*arrayname*]...

Remarks:

arrayname is the name of the array you want to erase.

You may want to use the ERASE statement if you are running short of storage space while running a program. After arrays are erased, the space in memory allocated for the arrays can be used for other purposes.

ERASE can also be used when you want to redimension arrays in your program. If you try to redimension an array without first erasing it, a **Duplicate definition** error occurs.

The CLEAR command erases *all* variables from the work area.

ERASE
Statement

Example: This example uses the FRE function to show how ERASE can be used to free memory. The array BIG used up about 4ØK bytes of memory (628Ø8-2198Ø) when it was dimensioned as BIG(1ØØØ). After it was erased, it could be redimensioned to BIG(1Ø,1Ø), and it took up only a little more than 5ØØ bytes (628Ø8-6228).

The actual values returned by the FRE function may be different on your computer.

```
1Ø START=FRE("")
2Ø DIM BIG(1ØØ,1ØØ)
3Ø MIDDLE=FRE("")
4Ø ERASE BIG
5Ø DIM BIG(1Ø,1Ø)
6Ø FINAL=FRE("")
7Ø PRINT START, MIDDLE, FINAL
RUN
 628Ø8          2198Ø          62289
```

ERDEV and ERDEV$ Variables

Purpose: Read-only variables. Hold the INTerrupt 24 error code of a device error, and the name of the device generating the error.

Not valid for BASIC releases earlier than 3.Ø.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	

Format: *v* = ERDEV

*v$* = ERDEV$

Remarks: ERDEV is a read-only variable. When an error in DOS is detected, ERDEV holds the INTerrupt 24 error code in the lower 8 bits, and the upper 8 bits contain bits 13, 14, and 15 of the attribute word of the device header block.

ERDEV$ is a read-only variable. If the error was on a character device, ERDEV$ contains the 8-byte character device name. If the error was not on a character device, ERDEV$ contains the two-character block device name (A:, B:, C:, etc.).

ERDEV and ERDEV$ Variables

Example: Open the B drive door and enter the following:

```
FILES "B:"
```

and BASIC returns:

```
Disk not ready
```

Then enter:

```
PRINT ERDEV, ERDEV$
```

and BASIC returns:

```
2            B:
```

Note: If you refer to *Disk Operating System Technical Reference* manual under the INT24 error code listing, you can see that error 2 is **Drive not ready.** The high-order 8 bits (the word attribute bits) are all zeros. As explained in the DOS *Technical Reference* section called "Attribute Field" under "Installable Device Drivers," bits 13, 14, and 15 set to zero means that B: is a block device, IOCTL is not supported, and the device is in IBM format.

See also "IOCTL Statement" and "IOCTL$ Function."

ERDEV and ERDEV$ Variables

This example simulates a printer error.

```
10 CLS
20 ON ERROR GOTO 50
30 LPRINT"The printer is ready"
40 PRINT"The printer is ready"
50 END
60 V$=HEX$(ERDEV)
70 PRINT "ERDEV = ";V$
80 D$=ERDEV$
90 PRINT "ERDEV$ = ";D$
100 RESUME NEXT
```

If you run this example with the printer turned off, the computer displays:

```
ERDEV = 8009
ERDEV$ = LPT1
```

The lower 8 bits (bits Ø-7) of the binary equivalent equal 9, which is the INT24 error code for **Printer out of paper**. The meaning of bits 13, 14, and 15 of the value returned by ERDEV is explained in the section "Attribute Field" of *Disk Operating System Technical Reference* under "Installable Device Drivers."

ERR and ERL Variables

Purpose: Return the error code and line number associated with an error.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = ERR

v = ERL

Remarks: The variable ERR contains the error code for the last error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF–THEN statements to direct program flow in the error-handling routine. See "ON ERROR Statement."

If you do test ERL in an IF–THEN statement, be sure to put the line number on the right side of the relational operator, like this:

```
IF ERL = line number THEN ...
```

The number must be on the right side of the operator to be renumbered by RENUM.

If the statement that caused the error was a direct mode statement, ERL contains 65535. You do not want this number changed during a RENUM, so to test whether an error occurred in a direct mode statement use the form:

```
IF 65535 = ERL THEN ...
```

ERR and ERL Variables

ERR and ERL can be set using the ERROR statement (see next entry).

BASIC error codes are listed in Appendix A, "Error Messages."

Example: This example tests to see if the drive door is open when the program needs to open a file.

```
10 ON ERROR GOTO 100
20 OPEN "DATA" FOR INPUT AS #1
30 END
   .
   .
   .
100 IF ERR=71 THEN LOCATE 23,1:
    PRINT "DISK IS NOT READY":RESUME
```

ERROR Statement

Purpose: Simulates the occurrence of a BASIC error; or allows you to define your own error codes.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: ERROR *n*

Remarks:

n must be an integer expression between Ø and 255.

If the value of *n* is the same as an error code used by BASIC (see Appendix A, "Error Messages"), the ERROR statement simulates the occurrence of that error. If an error-handling routine has been defined by the ON ERROR statement, the error routine is entered. Otherwise, the error message corresponding to the code is displayed, and execution halts. See the first example below.

To define your own error code, use a value that is different from any used by BASIC. (We suggest you use the highest available values; for example, values greater than 2ØØ.) This new error code can then be tested in an error handling routine, just like any other error. See the second example below.

If you define your own code in this way, and you don't handle it in an error handling routine, BASIC displays the message **Unprintable error**, and execution halts.

Example: The first example simulates a **String too long** error.

```
1Ø T = 15
2Ø ERROR T
RUN
String too long in line 2Ø
```

The next example is a part of a game program that allows you to make bets. By using an error code of 21Ø, which BASIC doesn't use, the program traps the error if you exceed the house limit.

```
1ØØ ON ERROR GOTO 1ØØØ
11Ø INPUT  "WHAT IS YOUR BET";B
12Ø IF B > 5ØØØ THEN ERROR 21Ø
   .
   .
   .
1ØØØ IF ERR = 21Ø THEN PRINT
     "HOUSE LIMIT IS $5ØØØ"
1Ø1Ø IF ERL = 12Ø THEN RESUME 11Ø
```

EXP
Function

Purpose: Calculates the exponential function.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: $v = \text{EXP}(x)$

Remarks: x can be any numeric expression.

This function returns the mathematical number e raised to the x power. e is the base for natural logarithms. An overflow occurs if x is greater than 88.Ø2969.

In BASIC 2.Ø and later releases, you can have this calculation performed in double-precision by specifying **/D** in the BASIC command line when BASIC is initially loaded. See "Options in the BASIC Command" in the *BASIC Handbook*.

Example: This example calculates e raised to the (2-1) power, which is simply e.

```
1Ø X = 2
2Ø PRINT EXP(X-1)
RUN
 2.718282
```

FIELD Statement

Purpose: Allocates space for variables in a random file buffer.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: FIELD [#]*filenum, width* AS *stringvar* [,*width* AS *stringvar*]...

Remarks:

filenum is the number under which the file was opened.

width is a numeric expression specifying the number of character positions to be allocated to *stringvar*.

stringvar is a string variable that is used for random file access.

A FIELD statement defines variables used to get data out of a random buffer after a GET or to enter data into the buffer for a PUT.

The statement:

```
FIELD 1, 2Ø AS N$, 1Ø AS ID$, 4Ø AS ADD$
```

allocates the first 2Ø positions (bytes) in the random file buffer to the string variable N$, the next 1Ø positions to ID$, and the next 4Ø positions to ADD$. FIELD does *not* actually place any data into the random file buffer. This is done by the LSET and RSET statements. See "LSET and RSET Statements."

FIELD Statement

FIELD does not "remove" data from the file either. Information is read from the file into the random file buffer with the GET (file) statement. Information is read from the buffer by simply referring to the variables defined in the FIELD statement.

The total number of bytes allocated in a FIELD statement must not exceed the record length specified when the file was opened. Otherwise, a **Field overflow** error occurs.

Any number of FIELD statements can be executed for the same file number, and all FIELD statements that have been executed are in effect at the same time. Each new FIELD statement redefines the buffer from the first character position, so this has the effect of having multiple field definitions for the same data.

> **Note:** Be careful about using a fielded variable name in an input or assignment statement. Once a variable name is defined in a FIELD statement, it points to the correct place in the random file buffer. If a subsequent input statement or LET statement with that variable name on the left side of the equal sign is executed, the variable is moved to string space and is no longer in the file buffer.

See Appendix A, "BASIC Disk Input and Output," in the *BASIC Handbook* for a complete explanation of how to use random files.

Example:

This example opens a file named "CUST" as a random file. The variable CUSTNO$ is assigned to the first two positions in each record; CUSTNAME$ is assigned to the next 3Ø positions; and ADDR$ is assigned to the next 35 positions.

Lines 3Ø through 5Ø put information into the buffer, and the PUT statement in line 6Ø writes the buffer to the file. Line 7Ø reads back that same record, and line 9Ø displays the three fields. Note in line 8Ø that it is permissible to use a variable name that was defined in a FIELD statement on the *right* side of an assignment statement.

```
1Ø OPEN  "A:CUST" AS #1
2Ø FIELD 1, 2 AS CUSTNO$, 3Ø AS CUSTNAME$,
      35 AS ADDR$
3Ø LSET CUSTNAME$= "O'NEIL INC"
4Ø LSET ADDR$= "5Ø SE 12TH ST, NY, NY"
5Ø LSET CUSTNO$=MKI$(785Ø)
6Ø PUT 1,1
7Ø GET 1,1
8Ø CNUM%= CVI(CUSTNO$): N$ = CUSTNAME$
9Ø PRINT CNUM%, N$, ADDR$
```

The program below shows a way to create a random file buffer with multiple FIELD statements in a line.

```
1Ø OPEN  "FOO" AS #1
2Ø FIELD 1, 1ØØ AS A$, 2ØØ AS B$
3Ø FIELD 1, 3ØØ AS DUMMY$, 4Ø AS C$
```

Note that in line 3Ø DUMMY$ moves the pointer into the file buffer so that you do not lose the information in line 2Ø.

FILES
Command

Purpose: Displays the names of files residing on the current directory of a disk. The FILES command in BASIC is similar to the DIR command in DOS.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	(**)

Format: FILES [*filespec*]

Remarks:

filespec is a string expression for the file specification. It must conform to the rules outlined under "Naming Files" in Chapter 3 of the *BASIC Handbook*; otherwise, a **Bad file name** error occurs. If *filespec* is omitted, all the files on the current directory of the DOS default drive are listed.

All files matching the filename are displayed. The filename can contain question marks (?). A question mark matches any character in the name or extension. An asterisk (*) as the first character of the name or extension matches any name or any extension.

If a drive is specified as part of *filespec*, files that match the specified filename on the current directory of that drive are listed. Otherwise, the DOS default drive is used.

Example: This command displays all files on the current directory of the DOS default drive.

```
FILES
```

This displays all files with an extension of .BAS on the current directory of the DOS default drive.

```
FILES  "*.BAS"
```

This displays all files on drive B.

```
FILES  "B:*.*"
```

This lists each file on the current directory of the DOS default drive that has a filename beginning with TEST followed by up to two other characters, and an extension of .BAS.

```
FILES  "TEST??.BAS"
```

Another way to list all the files on the current directory of drive B is: (For BASIC 2.Ø and later releases.)

```
FILES  "B:"
```

In addition to listing all the files on the current directory of the drive, BASIC displays the current directory name and the number of bytes free.

When using tree-structured directories, remember that each subdirectory contains two special entries. They are listed when you use the FILES command to list a subdirectory. The first contains a single period instead of a filename. It identifies this "file" as a subdirectory. The second entry contains two periods

instead of a filename. It is used to locate the higher level directory that defines this subdirectory. (For BASIC 2.Ø and later releases.)

This example lists all files in the current subdirectory called LEVEL1 on drive A. Note that the directory is empty.

```
FILES "A:\LEVEL1"

        .   <DIR>        ..  <DIR>

  32824 Bytes free
```

The FILES command can also be used to list files in other directories. The example below lists all files in the subdirectory LVL1. The backslash must be used after the directory name.

```
FILES  "LVL1\"
```

This example lists all files in the directory LVL2 with an extension of .BAS.

```
FILES  "LVL2\*.BAS"
```

FIX
Function

Purpose: Truncates *x* to an integer.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = FIX(*x*)

Remarks: *x* can be any numeric expression.

FIX strips all digits to the right of the decimal point and returns the value of the digits to the left of the decimal point.

The difference between FIX and INT is that FIX does not return the next lower number when *x* is negative.

See the "INT" and "CINT" functions, which also return integers.

Example: Note in the examples how FIX does *not* round the decimal part when it converts to an integer.

```
PRINT FIX(45.67)
 45

PRINT FIX(-2.89)
-2
```

FOR and NEXT Statements

Purpose: Performs a series of instructions in a loop a given number of times.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format:

FOR *variable*=*x* TO *y* [STEP *z*]

.

.

.

NEXT [*variable* [,*variable*]...]

Remarks:

variable is an integer or single-precision variable to be used as a counter.

x is a numeric expression that is the initial value of the counter.

y is a numeric expression that is the final value of the counter.

z is a numeric expression to be used as an increment.

The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by the STEP value (*z*). If you do not specify a value for *z*, the increment is assumed to be 1. A check is performed to see if the value of the counter is now greater than the final value *y*. If it is not greater, BASIC branches back to the statement after the FOR statement and the process is repeated.

If it is greater, execution continues with the statement following the NEXT statement. This is a FOR–NEXT loop.

If the value of *z* is negative, the test is reversed. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if *x* is already greater than *y* when the STEP value is positive, or *x* is less than *y* when the STEP value is negative. If *z* is zero, an infinite loop is created unless you provide some way to set the counter greater than the final value.

Program performance will be improved if you use integer counters whenever possible.

Nested Loops

FOR–NEXT loops can be nested; that is, one FOR–NEXT loop can be placed inside another FOR–NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement can be used for all of them.

FOR and NEXT Statements

A NEXT statement of the form:

```
NEXT var1, var2, var3 ...
```

is equivalent to the sequence of statements:

```
NEXT var1
NEXT var2
NEXT var3
    .
    .
    .
```

The variable(s) in the NEXT statement can be omitted, in which case the NEXT statement matches the most recent FOR statement. It is a good idea always to include the variables to avoid confusion; but it can be necessary if you do any branching out of nested loops. However, using variable names on the NEXT statements causes your program to execute somewhat slower.

Active loops should be exited by setting the loop counter out of range or setting a conditional statement with the loop causing the loop to terminate, so that every iteration of the FOR statement in the loop has a corresponding NEXT.

If a NEXT statement is encountered before its corresponding FOR statement, a **NEXT without FOR** error occurs.

Example: The first example shows a FOR–NEXT loop with a STEP value of 2.

```
10 J=10: K=30
20 FOR I=1 TO J STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
 1  40
 3  50
 5  60
 7  70
 9  80
```

In the following example, the loop does not execute because the initial value of the loop is more than the final value:

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
RUN
```

The next program results in a **NEXT without FOR** error. There can be only one NEXT statement for every FOR statement. (This is different from other versions of BASIC that allow a different physical NEXT statement when jumping out of a loop.)

```
10 FOR I=1 TO 5
20 IF I=2 GOTO 50
30 NEXT
40 GOTO 60
50 NEXT
60 END
```

FOR and NEXT Statements

In the last example, the loop executes 1Ø times. The final value for the loop variable is always set before the initial value is set. (This is different from some other versions of BASIC, which set the initial value of the counter before setting the final value. In another BASIC the loop in this example might execute six times.)

```
1Ø I=5
2Ø FOR I=1 TO I+5
3Ø PRINT I;
4Ø NEXT
RUN
 1  2  3  4  5  6  7  8  9  1Ø
```

FRE
Function

Purpose: Returns the number of bytes within BASIC's data space that are not being used. This number does not include the size of the reserved portion of the interpreter work area.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: v = FRE(x)

v = FRE(x\$)

Remarks: x and x\$ are dummy arguments.

Since strings in BASIC can have variable lengths (each time you do an assignment to a string its length can change), strings are manipulated dynamically. For this reason, string space can become fragmented, causing a decrease in program performance.

You can improve the performance of programs that execute many string functions by using the MID$ statement to access substrings imbedded within one large string. This prevents fragmentation of string space. See "MID$ Statement" for an example.

FRE with any string value causes a housecleaning before returning the number of free bytes. During *housecleaning*, BASIC collects all its useful data and frees up unused areas of memory once used for strings. The data is compressed so you can continue until you really run out of space.

FRE
Function

BASIC also automatically does a housecleaning when it is running out of usable work area. Be patient; housecleaning can take a while.

> **Note:** Ctrl-Break cannot be used during housecleaning.

Example: The actual value returned by FRE on your computer can differ from this example.

```
PRINT FRE(Ø)
 14542
```

GET
Statement (Files)

Purpose: Reads a record from a random file into a random buffer.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: GET [#]*filenum*[, *number*]

Remarks:

filenum is the number under which the file was opened.

number is the number of the record to be read, in the range 1 to 16 megabytes. If *number* is omitted, the next record (after the last GET) is read into the buffer.

After a GET statement, INPUT #, LINE INPUT #, or references to variables defined in the FIELD statement can be used to read characters from the random file buffer. See Appendix A, "BASIC Disk Input and Output," in the *BASIC Handbook* for more information on using GET.

Because BASIC and DOS block as many records as possible in 512-byte sectors, the GET statement does not necessarily perform a physical read from the disk.

GET can also be used for communications files. In this case *number* is the number of bytes to read from the communications buffer. This number cannot exceed the value set by the LEN option on the OPEN "COM... statement.

GET
Statement (Files)

Example: This example opens the file "CUST" for random access, with fields defined in line 2Ø. The GET statement on line 3Ø reads a record into the file buffer. Line 4Ø displays the information from the record that was read.

```
1Ø OPEN  "A:CUST" AS #1
2Ø FIELD 1, 3Ø AS CUSTNAME$, 3Ø AS ADDR$,
       35 AS CITY$
3Ø GET 1
4Ø PRINT CUSTNAME$, ADDR$, CITY$
```

GET
Statement (Graphics)

Purpose: Reads points from an area of the screen.

Versions:

Cassette	Disk	Advanced	Compiler
		***	***

Graphics mode only.

Format: GET (*x1,y1*)-(*x2,y2*),*arrayname*

Remarks:

(x1,y1), (x2,y2)
are coordinates in either absolute or relative form. Refer to "Specifying Coordinates" under "Graphics Modes" in Chapter 3 of the *BASIC Handbook* for more information on coordinates.

arrayname is the name of the array you want to hold the information.

GET reads the attributes of the points within the specified rectangle into the array. The specified rectangle has points (*x1,y1*) and (*x2,y2*) as opposite corners. (This is the same as the rectangle drawn by the LINE statement using the **B** option.)

GET and PUT can be used for high-speed object motion in graphics mode. You might think of GET and PUT as "bit pump" operations that move bits onto (PUT) and off (GET) the screen. Remember that PUT and GET are also used for random access files, but the syntax of these statements is different.

GET
Statement (Graphics)

The array is used simply as a place to hold the image and must be numeric; it can be any precision, however. The required size of the array, in bytes, is:

```
4+INT((x*bitsperpixel+7)/8)*y
```

where *x* and *y* are the lengths of the horizontal and vertical sides of the rectangle, respectively. The value of *bitsperpixel* is 2 in medium resolution, and 1 in high resolution.

For example, suppose you want to use the GET statement to get a 1Ø by 12 image in medium resolution. The number of bytes required is 4+INT((1Ø*2+7)/8)*12, or 4Ø bytes. The bytes per element of an array are:

- 2 for integer string
- 4 for single-precision string
- 8 for double-precision string

Therefore, you could use an integer array with at least 2Ø elements.

The information from the screen is stored in the array as follows:

1. 2 bytes giving the *x* dimension in bits
2. 2 bytes giving the *y* dimension in bits
3. the data itself

It is possible to examine the *x* and *y* dimensions and even the data itself if an integer array is used. The *x* dimension is in element Ø of the array, and the *y* dimension is in element 1.

Keep in mind that integers are stored low byte first, then high byte; but the data is actually transferred high byte first, then low byte.

The data for each row of points in the rectangle is left-justified on a byte boundary, so if less than a multiple of 8 bits is stored, the rest of the byte is filled with zeros.

PUT and GET work significantly faster in medium resolution when *x1* MOD 4 is equal to zero, and in high resolution when *x1* MOD 8 is equal to zero. This is a special case where the rectangle boundaries fall on the byte boundaries.

Example: See "PUT Statement (Graphics)" for an example.

GOSUB and RETURN Statements

Purpose: Branches to and returns from a subroutine.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format:

```
GOSUB line
  .
  .
  .
RETURN
```

Remarks:

line is the line number of the first line of the subroutine.

A subroutine can be called any number of times in a program, and a subroutine can be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement causes BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine can contain more than one RETURN statement, so you can return from different points in the subroutine. Subroutines can appear anywhere in the program.

To prevent your program from accidentally entering a subroutine, you can put a STOP, END, or GOTO statement before the subroutine to direct program control around it.

Use ON–GOSUB to branch to different subroutines based on the result of an expression.

GOSUB and RETURN Statements

Example: This example shows how a subroutine works. The GOSUB in line 1Ø calls the subroutine in line 4Ø. So the program branches to line 4Ø and starts executing statements there until it sees the RETURN statement in line 7Ø. At that point the program goes back to the statement after the subroutine call; that is, it returns to line 2Ø. The END statement in line 3Ø prevents the subroutine from being performed a second time.

```
1Ø GOSUB 4Ø
2Ø PRINT  "BACK FROM SUBROUTINE"
3Ø END
4Ø PRINT  "SUBROUTINE";
5Ø PRINT  " IN";
6Ø PRINT  " PROGRESS"
7Ø RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
```

GOTO
Statement

Purpose: Branches unconditionally out of the normal program sequence to a specified line number.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: GOTO *line*

Remarks:

line is the line number of a line in the program.

If *line* is the line number of an executable statement, that statement and those following are executed. If *line* refers to a nonexecutable statement (such as REM or DATA), the program continues at the first executable statement encountered after *line*.

The GOTO statement can be used in direct mode to reenter a program at a desired point. This can be useful in debugging.

Use ON–GOTO to branch to different lines based on the result of an expression.

GOTO Statement

Example: In this example, the GOTO statement in line 6Ø puts the program into an infinite loop, which is stopped when the program runs out of data in the DATA statement. (Notice how branching to the DATA statement does not add additional values to the internal data table.)

```
10 DATA 5,7,12
2Ø READ R
3Ø PRINT  "R = ";R,
4Ø A = 3.14*R^2
5Ø PRINT  "AREA = ";A
6Ø GOTO 10
RUN
R = 5           AREA = 78.5
R = 7           AREA = 153.86
R = 12          AREA = 452.16
OUT OF DATA IN 2Ø
```

HEX$ Function

Purpose: Returns a string that represents the hexadecimal value of the decimal argument.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = HEX$(*n*)

Remarks:

n is a numeric expression in the range -32768 to 65535.

If *n* is negative, the twos complement form is used. That is, HEX$(-*n*) is the same as HEX$(65536-*n*).

See "OCT$ Function" for octal conversion.

Example: The following example uses the HEX$ function to figure the hexadecimal representation for the two decimal values that are entered.

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X " DECIMAL IS ";A$ " HEXADECIMAL"
   RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
RUN
? 1023
 1023 DECIMAL IS 3FF HEXADECIMAL
```

IF
Statement

Purpose: Makes a decision regarding program flow based on the result of an expression.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: IF *expression* [,]THEN *clause* [ELSE *clause*]

IF *expression* [,]GOTO *line* [[,]ELSE *clause*]

Remarks:

expression can be any numeric expression.

clause can be a BASIC statement or a sequence of statements (separated by colons); or it can be simply the number of a line to branch to.

line is the line number of a line existing in the program.

If the *expression* is true (not zero), the THEN or GOTO clause is executed. THEN is followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number.

If the result of *expression* is false (zero), the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution then continues with the next numbered line containing an executable statement.

IF Statement

If you enter an IF–THEN statement in direct mode, and it directs control to a line number, an **Undefined line number** error results unless you previously entered a line with the specified line number.

> **Note:** When using IF to test equality for a value that is the result of a single- or double-precision computation, remember that the internal representation of the value may not be exact. (This is because single- and double-precision values are stored internally in floating point binary format.) Therefore, the test should be against the *range* over which the accuracy of the value can vary. For example, to test a computed variable A against the value 1.Ø, use:
>
> ```
> IF ABS (A-1.Ø)<1.ØE-6 THEN ...
> ```
>
> This test returns a true result if the value of A is 1.Ø with a relative error of less than 1.ØE-6.

Also note that IF–THEN–ELSE is just *one statement*. Once an IF statement occurs on a line, everything else on that line is part of the IF statement. Because IF–THEN–ELSE is all one statement, the ELSE clause cannot be a separate program line. For example:

```
1Ø IF A=B THEN X=4
2Ø ELSE P=Q
```

is invalid. Instead, it should be:

```
1Ø IF A=B THEN X=4 ELSE P=Q
```

Nesting of IF Statements: IF–THEN–ELSE statements can be nested. Nesting is limited only by the length of the line. For example,

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X
   THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a valid statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example:

```
IF A=B THEN IF B=C THEN PRINT "A=C"
   ELSE PRINT "A<>C"
```

will not print "A<>C" when A<>B.

Example: This statement gets record I if I is not zero:

```
100 IF I THEN GET #1,I
```

In the next example, if I is between 10 and 20, DB is calculated and execution branches to line 300. If I is not in this range, the message **Out of range** is printed. Note the use of two statements in the THEN clause.

```
100 IF (I>10) AND (I<20) THEN
       DB=1982-I: GOTO 300
    ELSE PRINT "OUT OF RANGE"
```

IF Statement

In the next example, in line 2Ø everything after the THEN is part of the clause. This means that the NEXT is not executed unless N=I. When line 2Ø executes, N does not equal I so the IF evaluation is false. Therefore, the NEXT is not performed and the program falls through to line 3Ø. The NEXT must be coded on a separate line if you want the program to loop until N=I.

```
1Ø N=15
2Ø FOR I=1 TO 2Ø:IF N=I THEN 4Ø:NEXT
3Ø PRINT "N <> I":END
4Ø PRINT "N = I"
RUN
N <> I
```

INKEY$
Variable

Purpose: Reads a character from the keyboard.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = INKEY$

Remarks: INKEY$ reads only a single character, even if several characters are waiting in the keyboard buffer. The returned value is a zero-, one-, or two-character string.

- A null string (length zero) indicates that no character is pending at the keyboard.
- A one-character string contains the actual character read from the keyboard.
- A two-character string indicates a special extended code. The first character is hex ØØ. For a complete list of these codes, see Appendix D, "ASCII Character Codes."

You must assign the result of INKEY$ to a string variable before using the character with any BASIC statement or function.

While INKEY$ is being used, no characters are displayed on the screen and all characters are passed through to the program except for:

- Ctrl-Break, which stops the program
- Ctrl-Num Lock, which sends the system into a pause state
- Alt-Ctrl-Del, which does a System Reset
- PrtSc, which prints the screen

INKEY$ Variable

If you press Enter in response to INKEY$, the carriage return character passes through to the program.

Example: The following section of a program stops execution until any key is pressed:

```
100 PRINT "Press any key to continue"
110 A$=INKEY$: IF A$="" THEN 110
```

The next example shows program lines that could be used to test a two-character code being returned:

```
100 KB$=INKEY$
110 IF LEN(KB$)=2 THEN KB$=RIGHT$(KB$,1)
```

INP
Function

Purpose: Returns the byte read from port *n*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: v = INP(n)

Remarks: *n* must be in the range Ø to 65535.

INP is the complementary function to the OUT statement. See "OUT Statement."

INP performs the same function as the IN instruction in assembly language. See also the IBM Personal Computer *Technical Reference* manual for a description of valid port numbers (I/O addresses).

Example: This instruction reads a byte from port 255 and assigns it to the variable A.

```
1ØØ A=INP(255)
```

INPUT
Statement

Purpose: Receives input from the keyboard during program execution.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: INPUT[;][*"prompt"*;] *variable*[,*variable*]...

Remarks:

"prompt" is a string constant that prompts for the desired input.

variable is the name of the numeric or string variable or array element that receives the input.

When the program sees an INPUT statement, it pauses and displays a question mark on the screen to indicate that it is waiting for data. If a "*prompt*" is included, the string is displayed. If "*prompt*" is followed by the semicolon (;), a question mark will follow the displayed string; if "*prompt* " is followed by a comma, the question mark is not displayed.

You can use a comma instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE",B$ prints the prompt without the question mark.

After the prompt or question mark is displayed, you can enter the required data from the keyboard. The data you enter is assigned to the *variables* given in the variable list. The data items you supply must be

separated by commas, and the number of data items must be the same as the number of variables in the list.

The type of data item that you enter must agree with the type specified by the variable name. (Strings entered in response to an INPUT statement need not be surrounded by quotation marks.)

If you respond to INPUT with too many or too few items, or with the wrong type of value (letters instead of numbers, etc.), BASIC displays the message **?Redo from start**. If a single variable is requested, you can simply press Enter to indicate the default values of Ø for numeric input or null for string input. However, if more than one variable is requested, pressing Enter causes the **?Redo from start** message to be printed because too few items were entered. BASIC does not assign any of the input values to variables until you give an acceptable response.

In Disk BASIC and Advanced BASIC, if INPUT is immediately followed by a semicolon, then pressing Enter to input data does not produce a carriage return/line feed sequence on the screen. This means that the cursor remains on the same line as your response.

Example: In this example, the question mark displayed by the computer is a prompt to tell you it wants you to enter something. Suppose you enter a 5.

```
1Ø INPUT X
2Ø PRINT X "SQUARED IS" X^2
3Ø END
RUN
?
```

The program continues:

```
    .
    .
    .
?  5
 5 SQUARED IS 25
```

For this second example, a prompt was included in line 2Ø, so this time the computer prompts with "WHAT IS THE RADIUS?"

```
1Ø PI=3.14
2Ø INPUT "WHAT IS THE RADIUS";R
3Ø A=PI*R∧2
4Ø PRINT "THE AREA OF THE CIRCLE IS ";A
5Ø END
RUN
WHAT IS THE RADIUS?
```

Suppose you respond with 7.4. The program continues:

```
    .
    .
    .
WHAT IS THE RADIUS?  7.4
THE AREA OF THE CIRCLE IS 171.9464
```

INPUT
Statement

Purpose: Reads data items from a sequential device or file and assigns them to program variables.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: INPUT #*filenum, variable* [,*variable*]...

Remarks:

filenum is the number used when the file was opened for input.

variable is the name of a variable that will have an item in the file assigned to it. It can be a string or numeric variable, or an array element.

The sequential file can reside on disk or on cassette; it can be a sequential data stream from a communications adapter; or it can be the keyboard (KYBD:).

The type of data in the file must match the type specified by the variable name. Unlike INPUT, no question mark is displayed with INPUT #.

The data items in the file must appear just as they would if the data were being typed in response to an INPUT statement. With numeric values, the leading spaces, carriage returns, and line feeds are ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of the number. The number ends with a space, carriage return, line feed, or comma.

If BASIC is scanning the data for a string item, the leading spaces, carriage returns, and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of the string item. If this first character is a quotation mark ("), the string item consists of all characters read between the first quotation mark and the second. Thus, a quoted string cannot contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string. It ends after a comma, carriage return, or line feed – or after 255 characters have been read. If end of file is reached when a numeric or string item is being input, the item is canceled.

Example: See Appendix A, "BASIC Disk Input and Output," in the *BASIC Handbook*.

INPUT$ Function

Purpose: Returns a string of *n* characters, read from the keyboard or from file number *filenum*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = INPUT$(*n*[,[#]*filenum*])

Remarks:

n is the number of characters to be read from the file.

filenum is the file number used on the OPEN statement. If *filenum* is omitted, the keyboard is read.

If the keyboard is used for input, no characters are displayed on the screen. All characters (including control characters) are passed through except Ctrl-Break, which is used to interrupt the execution of the INPUT$ function. When responding to INPUT$ from the keyboard, it is not necessary to press Enter.

The INPUT$ function allows you to read characters from the keyboard that are significant to the BASIC Program Editor, such as Backspace (ASCII code 8). If you want to read these special characters, you should use INPUT$ or INKEY$ (*not* INPUT or LINE INPUT).

For communications files, the INPUT$ function is preferred over the INPUT # and LINE INPUT #

statements, since all ASCII characters can be significant in communications. See also Appendix C, "Communications."

Example: The following program lists the contents of a sequential file in hexadecimal.

```
10 OPEN "DATA" FOR INPUT AS #1
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

The next example reads a single character from the keyboard in response to a question.

```
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
```

INSTR Function

Purpose: Searches for the first occurrence of string *y$* in *x$* and returns the position at which the match is found. The optional offset *n* sets the position for starting the search in *x$*

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = INSTR([*n*,]*x$*,*y$*)

Remarks:

n is a numeric expression in the range 1 to 255.

x$, y$ can be string variables, string expressions, or string constants.

If *n*>LEN(*x$*), or if *x$* is null, or if *y$* cannot be found, INSTR returns Ø. If *y$* is null, INSTR returns *n* (or 1 if *n* is not specified).

If *n* is out of range, an **Illegal function call** error is returned.

Example: This example searches for the string "B" within the string "ABCDEB". When the string is searched from the beginning, "B" is found at position 2; when the search starts at position 4, "B" is found at position 6.

```
1Ø A$ = "ABCDEB"
2Ø B$ = "B"
3Ø PRINT INSTR(A$,B$);INSTR(4,A$,B$)
RUN
  2  6
```

INT
Function

Purpose: Returns the largest integer that is less than or equal to *x*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: $v = \text{INT}(x)$

Remarks: *x* is any numeric expression.

This is called the "floor" function in some other programming languages.

See also "FIX" and "CINT" functions. (They also return integer values.)

Example: This example shows how INT truncates positive integers, but rounds negative numbers upward (in a negative direction).

```
PRINT INT(45.67)
 45
PRINT INT(-2.89)
-3
```

IOCTL Statement

Purpose: Allows BASIC to send a control data string to a device driver anytime *after* the driver has been OPENed.

Not valid for BASIC releases earlier than 3.Ø.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	

Format: IOCTL [#]*filenum,string*

Remarks:

filenum is the filenumber for the device driver.

string is a string expression containing the control data.

BASIC's file I/O system allows you to create and install your own device drivers. The IOCTL statement and the IOCTL$ function send control data to and read data from your device driver.

An IOCTL command string can be up to 255 bytes long. Multiple commands within the string can be separated by semicolons:

```
"LF;PL66;LW132"
```

You define the content and format of the control data string. The possible commands are determined by the characteristics of the driver installed.

Example: Initially, character device drivers for LPT1:, LPT2:, and LPT3: are installed, but they can be replaced. If you install a driver called LPT1 to replace LPT1: and that driver is able to set page length, an IOCTL command string to set or change the page length might be:

```
"PLn" (where n is the new page length).
```

You can then open the new LPT1 driver and set the page length with:

```
OPEN "LPT1" FOR OUTPUT AS #1
IOCTL #1, "PL60"
```

You could, for instance, write a device driver that controls a monitor and is capable of setting the mode of the screen to color and also capable of setting the width of the screen. For example:

```
OPEN "OPT" FOR OUTPUT AS #2
IOCTL #2, "CL:W40"
```

Assuming that your new driver accepts a command called "CL" to change the screen to color and a command called "W*n*" to set the width of the screen, the previous example passes those commands to your driver and causes the screen to respond.

Note: For related information, see "IOCTL$ Function" in this manual, "Device Drivers" in the *BASIC Handbook*, and the device driver section of *Disk Operating System Technical Reference*.

IOCTL$ Function

Purpose: Reads a control data string from a device driver that is open.

Not valid for BASIC releases earlier than 3.Ø.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	

Format: *v$* = IOCTL$([#]*filenum*)

Remarks:

filenum is the number of the file open to the device.

The IOTCL$ function can be used to get acknowledgment that an IOCTL command has succeeded or failed. It can also be used to get device configuration information, such as device width.

Example: This example checks to see if control data was successfully received.

```
1Ø OPEN "COM" AS #1
2Ø IOCTL #1, "SW132;GW"
3Ø IF IOCTL$(1) = "132"
     THEN PRINT "WIDTH SET SUCCESSFULLY"
```

If the device driver "COM" returns a value not equal to 132 from the IOCTL$ request, your command was not processed successfully and you should check for errors. If a device failure occurs, check the system variables of ERDEV and ERDEV$.

KEY Statement

Purpose: Sets or displays the soft keys.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: KEY ON

KEY OFF

KEY LIST

KEY *n, x$*

KEY *n,* CHR$(KB*flag*) and CHR$(*scan code*) (For Advanced BASIC 2.0 and later releases.)

Remarks: KEY ON causes the soft key values to be displayed on the 25th line. When the width is 4Ø, 5 of the 1Ø soft keys are displayed. When the width is 8Ø, all 1Ø are displayed. In either width, only the first 6 characters of each value are displayed.

KEY OFF erases the soft key display from the 25th line, making that line available for program use. It does not disable the function keys.

After turning off the soft key display with KEY OFF, you can use LOCATE 25,1 followed by PRINT to display anything you want on the bottom line of the screen. Information on line 25 is not scrolled, as are lines 1 through 24.

KEY Statement

KEY LIST lists all 1Ø soft key values on the screen. All 15 characters of each value are displayed.

KEY *n,x$* allows you to set each function key to automatically type any sequence of characters. ON is the default state for the soft key display.

n is the function key number in the range 1 to 1Ø.

x$ is a string expression that is assigned to the key. (Remember to enclose string *constants* in quotation marks.)

The value of a function key *n* is reassigned the value of the string *x*$. If the value entered for *n* is not in the range 1 to 1Ø, an **Illegal function call** error occurs. The previous key string assignment is retained. *x$* can be 1 to 15 characters in length. If it is longer than 15 characters, only the first 15 characters are assigned.

Assigning a null string to a soft key disables the function key as a soft key.

When a soft key is pressed, the INKEY$ function returns one character of the soft key string each time it is called. The first character is binary zero, the second is the key scan code, as listed in Appendix D, "ASCII Character Codes."

There are also six definable key traps. With this capability, you can trap any Ctrl, Shift, or super-shift key. (For Advanced BASIC 2.Ø and later releases.)

These additional keys are defined by the statement:

```
KEY n,CHR$(KBflag)+CHR$(scan code)
```

n	is a numeric expression in the range 15 to 2Ø.
KBflag	is a mask for the latched keys. The appropriate bit in *KBflag* must be set in order to trap a key that is shifted, Alt-shifted, or Ctrl-shifted. The KBflag values in hex are :
Caps Lock	&HØ–Caps Lock is not active.
Caps Lock	&H4Ø–Caps Lock is active.
Num Lock	&HØ–Num Lock is not active.
Num Lock	&H2Ø–Num Lock is active.
Alt	&HØ8–ALT key is pressed.
Ctrl	&HØ4–Control key is pressed.
Left Shift	&HØ2–Left Shift key is pressed.
Right Shift	&HØ1–Right Shift is pressed.
Scan code	is the number identifying one of the 83 keys to trap. See Appendix E, "Scan Codes."

Note that key trapping assumes that the left and right Shift keys are the same, so you can use a value of &HØ1, &HØ2, or &HØ3 (the sum of hex Ø1 and hex Ø2) to denote a Shift key.

You can also add multiple shift states. For example, the Ctrl and Alt keys can be added together. Shift state values *must* be in hex.

When trapping a key or key combinations, you must know the state of *Num Lock* and *Caps Lock*.

When you trap keys, they are processed in the following order:

1. Ctrl-PrtSc, which activates the line printer, is processed first. Even if Ctrl-PrtSc is defined as a trappable key, this does not prevent characters from being echoed to the line printer.

2. Next, the function keys F1 to F1Ø, Cursor Up, Cursor Down, Cursor Right, and Cursor Left (1-14) are processed. Setting scan codes 59 to 68, 72, 75, 77, or 8Ø as key traps has no effect, because they are considered to be predefined.

3. Last, the keys you define for 15 to 2Ø are processed.

Notes:

- Trapped keys do not go into the BIOS buffer so that only BASIC will know that the keys were pressed.

- Be careful when you trap Ctrl-Break and Ctrl-Alt-Del, because unless you have a test in your trap routine, you will have to turn the power off to stop your program.

KEY Statement

The following entry, "KEY(n) Statement," explains how to enable and disable function key trapping in Advanced BASIC.

Example: This example displays the soft keys on the 25th line.

```
50 KEY ON
```

This example erases soft-key display. The soft keys are still active, but not displayed.

```
10 KEY OFF
```

This example assigns the string "FILES"+Enter to soft key 1. This is a way to assign a commonly used command to a function key.

```
10 KEY 1,"FILES"+CHR$(13)
```

This example disables function key 1 as a soft key.

```
10 KEY 1,""
```

This example sets up a Key trap for capital P. Note that all three KEY statements – KEY, KEY(n), and ON KEY—are used with key trapping.

```
10 KEY 15, CHR$(&H40)+CHR$(25)
20 ON KEY(15) GOSUB 1000
30 KEY(15) ON
```

This example sets up a Key trap for Ctrl-Shift A. Notice that the hex values for Ctrl (&HØ4) and Shift (&HØ3) are added together to get the shift state.

```
10 KEY 20, CHR$(&H04+&H03)+CHR$(30)
20 ON KEY(20) GOSUB 2000
30 KEY(20) ON
```

KEY(n) Statement

Purpose: Activates and deactivates trapping of the specified key in a BASIC program. See "ON KEY(n) Statement."

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: KEY(*n*) ON

KEY(*n*) OFF

KEY(*n*) STOP

Remarks:

n is a numeric expression in the range 1 to 2Ø, and indicates the trapped key:

1-1Ø	function keys F1 to F1Ø
11	Cursor Up
12	Cursor Left
13	Cursor Right
14	Cursor Down
15-2Ø	keys defined by the form: KEY *n*,CHR$(*KBflag*)+CHR$(*scan code*). Keys 15-2Ø can be trapped in Advanced BASIC 2.Ø and later releases.

KEY(*n*) ON must be executed to activate trapping of function key or cursor control key activity. After KEY(*n*) ON, if a nonzero line number is specified in the ON KEY(*n*) statement, then every time BASIC starts a new statement it checks to see if the specified key was pressed. If so it performs a

GOSUB to the line number specified in the ON KEY(*n*) statement. A KEY(*n*) statement cannot precede an ON KEY(*n*) statement.

If KEY(*n*) is OFF, no trapping takes place and even if the key is pressed, the event is not remembered.

Once a KEY(*n*) STOP statement has been executed, no trapping takes place. However, if you press the specified key your action is remembered, so that an immediate trap takes place when KEY(*n*) ON is executed.

KEY(*n*) ON has no effect on whether the soft key values are displayed at the bottom of the screen.

If you use a KEY(n) statement in Cassette BASIC or Disk BASIC, a **Syntax error** occurs. See also "KEY Statement."

KILL Command

Purpose: Deletes a file from a disk. The KILL command in BASIC is similar to the ERASE command in DOS.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	

Format: KILL *filespec*

Remarks:

filespec is a string expression for the file specification. In BASIC 2.Ø and later releases, it can contain a path. *Filespec* must conform to the rules outlined under *Naming Files* in Chapter 3 of the *BASIC Handbook*; otherwise, an error occurs.

KILL can be used for all types of disk files. The name must include the extension, if one exists. For example, you can save a BASIC program using the command

```
SAVE "TEST"
```

BASIC supplies the extension **.BAS** for the SAVE command, but not for the KILL command. If you want to delete that program file later, you must say KILL "TEST.BAS", not KILL "TEST".

If a KILL statement is given for a file that is currently open, a **File already open** error occurs.

Example: To delete the file named "DATA1" on drive A, you might use:

```
200 KILL "A:DATA1"
```

To delete the file "PROG.BAS" in the LEVEL2 subdirectory, you might use:

```
KILL "LEVEL1\LEVEL2\PROG.BAS"
```

Note that KILL can be used only to delete files. The RMDIR command must be used to remove directories.

LEFT$
Function

Purpose: Returns the leftmost *n* characters of *x$*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = LEFT$(*x$*,*n*)

Remarks:

x$ is any string expression.

n is a numeric expression that must be in the range Ø to 255. It specifies the number of characters that are to be in the result.

If *n* is greater than LEN(*x$*), the entire string (*x$*) is returned. If *n*=Ø, the null string (length zero) is returned.

See also "MID$" and "RIGHT$" functions.

Example: In this example, the LEFT$ function is used to extract the first five characters from the string "BASIC PROGRAM".

```
1Ø A$ = "BASIC PROGRAM"
2Ø B$ = LEFT$(A$,5)
3Ø PRINT B$
RUN
BASIC
```

LEN
Function

Purpose: Returns the number of characters in *x$*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = LEN(*x$*)

Remarks: *x$* is any string expression.

Unprintable characters and blanks are included in the count of the number of characters.

Example: There are 14 characters in the string "BOCA RATON, FL" because the comma and the two blanks are counted.

```
10 X$ = "BOCA RATON, FL"
20 PRINT LEN(X$)
RUN
 14
```

LET Statement

Purpose: Assigns the value of an expression to a variable.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: [LET] *variable=expression*

Remarks:

variable is the name of the variable or array element that is to receive a value. It can be a string or numeric variable or array element.

expression is the expression whose value is assigned to *variable*. The type of the expression (string or numeric) must match the type of the variable, or a **Type mismatch** error occurs.

The word LET is optional; that is, the equal sign is enough when assigning an expression to a variable name.

Example: This example assigns the value 12 to the variable DORI. It then assigns the value 14, which is the value of the expression DORI+2, to the variable E. The string "HORA" is assigned to the variable FDANCE$.

```
10 LET DORI=12
20 LET E=DORI+2
30 LET FDANCE$="HORA"
```

The same statements could have also been written:

```
10 DORI=12
20 E=DORI+2
30 FDANCE$="HORA"
```

LINE Statement

Purpose: Draws a line or a box on the screen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Graphics mode only.

Format: LINE [(*x1,y1*)] -(*x2,y2*) [,[*color*] [,B[F]] [,*style*]]

Remarks:

(x1,y1), (x2,y2)
are coordinates in either absolute or relative form. See "Specifying Coordinates" under "Graphics Modes" in Chapter 3 of the *BASIC Handbook*.

color is an integer expression. It chooses an attribute from the legal attribute range for the current screen mode. In medium resolution, the color is the current color for that attribute as defined by the COLOR statement. Four attributes (Ø-3) are available in medium resolution; in high resolution, two attributes (Ø-1) are available. Zero is always the attribute for the background. The default foreground attribute is always the maximum attribute for that screen mode: 3 in medium resolution; 1 in high resolution.

style is a 16-bit integer mask used to put points on the screen. The *style* option is used for normal lines and boxes, but cannot be used with filled boxes (BF).

Using *style* with BF results in a **Syntax error**. This technique is called line styling. (For BASIC 2.Ø and later releases.)

The simplest form of LINE is:

```
LINE -(X2,Y2)
```

This will draw a line from the last point referenced to the point (*x2,y2*) in the foreground attribute.

We can include a starting point also:

```
LINE (Ø,Ø)-(319,199) 'diagonal down screen
LINE (Ø,1ØØ)-(319,1ØØ) 'horizontal bar
  across screen
```

We can indicate the attribute in which to draw the line:

```
LINE (1Ø,1Ø)-(2Ø,2Ø),2 'draw in attribute 2
```

```
1Ø 'draw random lines in random colors
2Ø SCREEN 1,Ø,Ø,Ø: CLS
3Ø LINE -(RND*319,RND*199),RND*4
4Ø GOTO 2Ø
```

```
1Ø 'alternating pattern - line on, line off
2Ø SCREEN 1,Ø,Ø,Ø: CLS
3Ø FOR X=Ø TO 319
4Ø LINE (X,Ø)-(X,199),X AND 1
5Ø NEXT
```

The next argument to LINE is **B** (box), or **BF** (filled box). We can leave out *color* and include the argument:

```
LINE (Ø,Ø)-(1ØØ,1ØØ),,B 'box in foreground
```

or we can include the attribute:

```
LINE (Ø,Ø)-(1ØØ,1ØØ),2,BF
   ' filled box attribute 2
```

The **B** tells BASIC to draw a rectangle with the points (*x1,y1*) and (*x2,y2*) as opposite corners. This avoids having to give the four LINE commands:

```
LINE (X1,Y1)-(X2,Y1)
LINE (X1,Y1)-(X1,Y2)
LINE (X2,Y1)-(X2,Y2)
LINE (X1,Y2)-(X2,Y2)
```

that perform the equivalent function.

The **BF** means "draw the same rectangle as **B**, but also fill in the interior points with the selected color."

The last argument to LINE is *style*. LINE uses the current circulating bit in *style* to plot (or store) points on the screen. If the bit is Ø (zero), no point is plotted. If the bit is 1 (one), a point is plotted. After each point, the next bit position in *style* is selected. When the last bit position has been selected, LINE "wraps around" and begins with the first bit again.

Note that a Ø (zero) bit indicates no store and does not erase the existing point on the screen. You may want to draw a background line before a styled line to force a known background.

The *style* option can be used to draw a dotted line across the screen by plotting (storing) every other point. Because *style* is 16 bits wide, the pattern for a dotted line looks like this:

```
1 Ø 1 Ø 1 Ø 1 Ø 1 Ø 1 Ø 1 Ø 1 Ø
```

This is equal to HAAAA in hexadecimal notation.

Examples: To draw a dotted line:

```
1Ø SCREEN 1,Ø
2Ø LINE (Ø,Ø)-(319,199),,,&HAAAA
```

To draw a cyan box with dashes:

```
1Ø SCREEN 1,Ø
2Ø LINE (Ø,Ø)-(1ØØ,1ØØ),1,B,&HCCCC
```

In BASIC release 1.1, out-of-range coordinates given to the LINE statement wrap-around to the next horizontal line.

In BASIC 2.Ø and later releases, out-of-range coordinates are clipped.

The last point referenced after a LINE statement is point (*x2,y2*). If you use the relative form for the second coordinate, it is relative to the first coordinate. For example,

```
LINE (1ØØ,1ØØ)-STEP (1Ø,-2Ø)
```

draws a line from (1ØØ,1ØØ) to (11Ø,8Ø).

This example draws random boxes filled with random colors.

```
1Ø CLS
2Ø SCREEN 1,Ø: COLOR Ø,Ø
3Ø LINE -(RND*319,RND*199),RND*2+1,BF
4Ø GOTO 3Ø 'boxes will overlap
```

LINE INPUT Statement

Purpose: Reads an entire line (up to 255 characters) from the keyboard into a string variable, ignoring delimiters.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: LINE INPUT[;][*"prompt"*;] *stringvar*

Remarks:

"prompt" is a string constant that is displayed on the screen before input is accepted. A question mark is not printed unless it is part of the prompt string.

stringvar is the name of the string variable or array element to which the line will be assigned. All input from the end of the prompt to the Enter is assigned to *stringvar*. Trailing blanks are ignored.

In Disk BASIC and Advanced BASIC, if LINE INPUT is immediately followed by a semicolon, then pressing Enter to end the input line does not produce a carriage return/line feed sequence on the screen. That is, the cursor remains on the same line as your response.

You can exit LINE INPUT by pressing Ctrl-Break. BASIC returns to command level and displays **Ok**. You can then enter CONT to resume execution at the LINE INPUT.

Example: See the example in the next entry, "LINE INPUT # Statement."

LINE INPUT#
Statement

Purpose: Reads an entire line (up to 255 characters), ignoring delimiters, from a sequential file into a string variable.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: LINE INPUT #*filenum, stringvar*

Remarks:

filenum is the number under which the file was opened.

stringvar is the name of a string variable or array element to which the line will be assigned.

LINE INPUT # reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence, and the next LINE INPUT # reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved. That is, the line feed/carriage return characters are returned as part of the string.)

LINE INPUT # is especially useful if each line of a file has been broken into fields, or if a BASIC program saved in ASCII mode is being read as data by another program.

See also Appendix A, "BASIC Disk Input and Output," in the *BASIC Handbook*.

LINE INPUT
Statement

Example: The following example uses LINE INPUT to get information from the keyboard, where the information is likely to have commas or other delimiters. Then the information is written to a sequential file, and read back out from the file using LINE INPUT #.

```
10 OPEN "LST" FOR OUTPUT AS #1
20 LINE INPUT "Address? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "LST" FOR INPUT AS #1
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN

Address?
```

Suppose you respond with DELRAY BEACH, FL 33445. The program continues:

```
.
.
.
Address? DELRAY BEACH, FL    33445

DELRAY BEACH, FL    33445
```

LIST Command

Purpose: Lists the program currently in memory on the screen or other specified device.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: LIST [*line1*] [-[*line2*]] [,*filespec*]

Remarks:

line1, line2 are valid line numbers in the range Ø to 65529. *line1* is the first line to be listed. *line2* is the last line to be listed. A period (.) can be used for either line number to indicate the current line.

filespec is a string expression for the file specification. In BASIC 2.Ø and later releases, it can contain a path. *Filespec* must conform to the rules outlined under "Naming Files" in Chapter 3 of the *BASIC Handbook*; otherwise, an error occurs.

In Cassette BASIC, listings directed to the screen by omitting the device specifier can be stopped at any time by pressing Ctrl-Break. Listings directed to specific devices cannot be interrupted and will list until the range is too big. That is, LIST *range* can be interrupted, but LIST *range*,"SCRN:" cannot.

In Disk BASIC and Advanced BASIC, any listing to either the screen or the printer can be interrupted by pressing Ctrl-Break.

If the line range is omitted, the entire program is listed.

When the dash (-) is used in a line range, three options are available:

- If only *line1* is given, that line and all higher numbered lines are listed.
- If only *line2* is given, all lines from the beginning of the program through *line2* are listed.
- If both line numbers are specified, all lines from *line1* through *line2*, inclusive, are listed.

When you list to a file on cassette or disk, the specified part of the program is saved in ASCII format. This file can later be used with MERGE.

BASIC always returns to the command level after a LIST is executed.

Example: This example lists the entire program on the screen.

```
LIST
```

This example lists line 35 on the screen.

```
LIST 35,"SCRN:"
```

This example lists lines 1Ø through 2Ø on the printer.

```
LIST 1Ø-2Ø, "LPT1:"
```

This example lists all lines from 1ØØ through the end of the program to the first communications adapter at 12ØØ bps, no parity, 8 data bits, 1 stop bit.

```
LIST 1ØØ- ,"COM1:12ØØ,N,8"
```

This example lists from the first line through line 2ØØ to a file named "BOB" on cassette.

```
LIST -2ØØ,"CAS1:BOB"
```

LLIST Command

Purpose: Lists all or part of the program currently in memory on the printer (LPT1:).

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: LLIST [*line1*][- [*line2*]]

Remarks: The line number ranges for LLIST work the same as for LIST.

BASIC always returns to command level after an LLIST is executed.

Example: This example prints the entire program.

```
LLIST
```

This example prints line 35.

```
LLIST 35
```

This example prints lines 1Ø through 2Ø.

```
LLIST 1Ø-2Ø
```

This example prints all lines from 1ØØ through the end of the program.

```
LLIST 1ØØ-
```

This example prints the first line through line 2ØØ.

```
LLIST -2ØØ
```

LOAD Command

Purpose: Loads a program from the specified device into memory, and optionally runs it.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: LOAD *filespec*[,R]

Remarks:

filespec is a string expression for the file specification. In BASIC 2.Ø and later releases, it can contain a path. *Filespec* must conform to the rules outlined under "Naming Files" in Chapter 3 of the *BASIC Handbook*; otherwise, an error occurs and the load is canceled.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the specified program. If the **R** option is omitted, BASIC returns to direct mode after the program is loaded.

However, if the **R** option is used with LOAD, the program runs after it is loaded. In this case all open data files are kept open. Thus, LOAD with the **R** option can be used to chain several programs (or segments of the same program). Information can be passed between the programs using data files.

LOAD *filespec*,R is equivalent to RUN *filespec*.

If you are using Cassette BASIC and the device name is omitted, CAS1: is assumed. CAS1: is the only device allowed for LOAD in Cassette BASIC.

If you are using Disk BASIC or Advanced BASIC, the DOS default disk drive is used if the device is omitted. The extension .BAS is added to the filename if no extension is supplied and the filename is eight characters or less.

Notes when using CAS1:

1. If the LOAD statement is entered in direct mode, the file names on the tape are displayed on the screen followed by a period (.) and a single letter indicating the type of file. This is followed by the message **Skipped** for the files not matching the named file, and **Found** when the named file is found. Types of files and their corresponding letter are:

   .B for BASIC programs in internal format (created with SAVE command)
   .P for protected BASIC programs in internal format (created with SAVE ,P command)
   .A for BASIC programs in ASCII format (created with SAVE ,A command)
   .M for memory image files (created with BSAVE command)
   .D for data files (created by OPEN followed by output statements)

   To see what files are on a cassette tape, rewind the tape and enter some name that is known not to be on the tape; for example, **LOAD "CAS1:NOWHERE"**. All filenames are then displayed.

   If the LOAD command is executed in a BASIC program, the filenames skipped and found are not displayed on the screen.

2. Note that Ctrl-Break can be typed at any time during LOAD. Between files or after a time-out period, BASIC exits the search and returns to command level. Previous memory contents remain unchanged.

3. If CAS1: is specified as the device and the filename is omitted, the next program file on the tape is loaded.

Example: This example loads the program named MENU, but does not run it.

```
LOAD "MENU"
```

This example loads and runs the program INVENT:

```
LOAD "INVENT",R
```

which is equivalent to

```
RUN "INVENT"
```

This example loads the file VLAD.BAS from disk drive B. Note that the .BAS did not have to be specified.

```
LOAD "B:VLAD.BAS"
```

This example loads the next program on the tape.

```
LOAD "CAS1:"
```

LOC
Function

Purpose: Returns the current position in the file.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: *v* = LOC(*filenum*)

Remarks:

filenum is the file number used when the file was opened.

With random files, LOC returns the record number of the last record read or written to a random file since the file was opened.

With sequential files, LOC returns the number of records read from or written to the file since it was opened. (A record for sequential files is a 128-byte block of data.) When a file is opened for sequential input, BASIC reads the first sector of the file, so LOC returns a 1 even before any input from the file.

For a communications file, LOC returns the number of characters in the input buffer waiting to be read. The default size for the input buffer is 256 characters, but you can change this with the **/C:** option in the BASIC command line. If more than 255 characters are in the buffer, LOC returns 255. Since a string is limited to 255 characters, this practical limit means that you do not have to test for string size before reading data into it. If fewer than 255 characters remain in the buffer, then LOC returns the actual count.

Example: This example stops the program when the 5Øth record in the file is passed.

```
1ØØ IF LOC(1)>5Ø THEN STOP
```

This example could be used to rewrite the record that was just read.

```
1ØØ PUT #1,LOC(1)
```

LOCATE Statement

Purpose: Positions the cursor on the active screen. Optional parameters turn the blinking cursor on and off and define the size of the blinking cursor.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: LOCATE [*row*][,[*col*][,[*cursor*][,[*start*][,*stop*]]]]

Remarks:

row is a numeric expression in the range 1 to 25. It indicates the screen line number where you want to place the cursor.

col is a numeric expression in the range 1 to 4Ø or 1 to 8Ø, depending upon screen width. It indicates the screen column number where you want to place the cursor.

cursor is a value indicating whether the cursor is visible or not. A Ø (zero) indicates off, 1 (one) indicates on.

start is the cursor-start scan line. It must be a numeric expression in the range Ø to 31.

stop is the cursor-stop scan line. It also must be numeric expression in the range Ø to 31.

cursor, start, and *stop* do not apply to graphics mode.

start and *stop* allow you to make the cursor any size you want. You indicate the starting and ending scan

lines. The scan lines are numbered from Ø at the top of the character position. The bottom scan line is 7 if you have the Color/Graphics Monitor Adapter, 13 if you have the IBM Monochrome Display and Parallel Printer Adapter. If *start* is given and *stop* is omitted, *stop* assumes the value of *start*. If *start* is greater than *stop*, you'll get a two-part cursor. The cursor "wraps" from the bottom line back to the top.

After a LOCATE statement, I/O statements to the screen begin placing characters at the specified location.

When a program is running, the cursor is normally off. You can use LOCATE ,,1 to turn it back on.

Normally, BASIC will not print to line 25. However, you can turn off the soft key display using KEY OFF, and then use LOCATE 25,1: PRINT... to put data on line 25. Line 25 does not scroll as the rest of the screen does.

Any parameter can be omitted. Omitted parameters assume the current value.

Any values entered outside the ranges indicated results in an **Illegal function call** error. Previous values are retained.

LOCATE
Statement

Example: This example moves the cursor to the home position in the upper left-hand corner of the screen.

```
10 LOCATE 1,1
```

This example makes the blinking cursor visible; its position remains unchanged.

```
10 LOCATE ,,1
```

In this example, position and cursor visibility remain unchanged. The cursor is set to display at the bottom of the character on the Color/Graphics Monitor Adapter (starting and ending on scan line 7).

```
10 LOCATE ,,,7
```

This example moves the cursor to line 5, column 1. It makes the cursor visible, covering the entire character cell on the Color/Graphics Monitor Adapter, starting at scan line Ø and ending on scan line 7.

```
10 LOCATE 5,1,1,Ø,7
```

LOF
Function

Purpose: Returns the number of bytes allocated to the file (length of the file).

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: *v* = LOF(*filenum*)

Remarks:

filenum is the file number used when the file was opened.

For disk files created by BASIC 1.1Ø, LOF returns a multiple of 128. For example, if the actual data in the file is 257 bytes, the number 384 is returned. For disk files created outside BASIC (for example, by using EDLIN), and for files created by BASIC 2.Ø and later releases, LOF returns the actual number of bytes allocated to the file.

For communications, LOF returns the amount of free space in the input buffer. That is, *size*-LOC(*filenum*), where *size* is the size of the communications buffer, which defaults to 256 but can be changed with the **/C:** option in the BASIC command line. Use of LOF can be used to detect when the input buffer is getting full. In practicality, LOC is adequate for this purpose, as demonstrated in the example in Appendix C, "Communications."

LOF
Function

Example: These statements get the last record of the file named BIG (if BIG was created with a record length of 128 bytes):

```
10 OPEN "BIG" AS #1
20 GET #1,LOF(1)/128
```

LOG
Function

Purpose: Returns the natural logarithm of x.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: $v = \text{LOG}(x)$

Remarks:

x must be a numeric expression greater than zero.

The natural logarithm is the logarithm to the base e. In BASIC 2.Ø and later releases, you can have this calculation performed in double-precision by specifying **/D** in the BASIC command line when BASIC is initially loaded. See "Options in the BASIC Command Line" in the *BASIC Handbook*.

Example: The first example calculates the logarithm of the expression 45/7:

```
PRINT LOG(45/7)
 1.86Ø752
```

The second example calculates the logarithm of e and of e^2:

```
E= 2.718282
? LOG(E)
 1
? LOG(E*E)
 2
```

LPOS
Function

Purpose: Returns the current position of the print head within the printer buffer for LPT1:.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: v = LPOS(n)

Remarks:

n is a numeric expression that is a dummy argument in Cassette BASIC. In Disk BASIC and Advanced BASIC, n indicates the printer being tested, as follows:

Ø or 1	LPT1:
2	LPT2:
3	LPT3:

Therefore, we recommend you use Ø or 1 in Cassette BASIC to maintain compatibility with the other versions.

The LPOS function does not necessarily give the physical position of the print head on the printer.

Example: In this example, if the line length is more than 6Ø characters a carriage return character is sent to the printer so it skips to the next line.

```
1ØØ IF LPOS(Ø)>6Ø THEN LPRINT CHR$(13)
```

LPRINT and LPRINT USING Statements

Purpose: Prints data on the printer (LPT1:).

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: LPRINT [*list of expressions* [;]

LPRINT USING *v$; list of expressions* [;]

Remarks:

list of expressions
is a list of the numeric and/or string expressions to be printed. The expressions must be separated by commas or semicolons.

expression is a numeric or string expression whose value is to be printed.

v$ is a string constant or variable that identifies the format to be used for printing. This is explained in detail under "PRINT Statement."

These statements function like PRINT and PRINT USING, except output goes to the printer. See "PRINT" and "PRINT USING" statements.

LPRINT assumes an 8Ø-character-wide printer. That is, BASIC automatically inserts a carriage return/line feed after printing 8Ø characters. This means that two lines are skipped when you print exactly 8Ø characters, unless you end the statement with a semicolon. You can change the width value with a WIDTH "LPT1:" statement.

If you do a form feed (LPRINT CHR$(12);) followed by another LPRINT and the printer takes more than 1Ø seconds to do the form feed, you can get a **Device timeout** error on the second LPRINT. To avoid this problem, enter the following:

```
1 ON ERROR GOTO 65ØØØ
    .
    .
    .
65ØØØ IF ERR = 24 THEN RESUME '24=timeout
```

You may want to test ERR to make sure the timeout was caused by an LPRINT statement.

Example: This is an example of sending special control characters to the IBM 8Ø CPS Matrix Printer using LPRINT and CHR$. The printer control characters are listed in the IBM Personal Computer *Technical Reference* manual.

```
1Ø LPRINT CHR$(14);"    Title Line"
2Ø FOR I=2 TO 4
3Ø LPRINT "Report line";I
4Ø NEXT I
5Ø LPRINT CHR$(15);"Condensed print;132
                    char/line"
6Ø LPRINT CHR$(18);"Return to normal"
7Ø LPRINT CHR$(27);"E"
8Ø LPRINT "This is emphasized print"
9Ø LPRINT CHR$(27);"F"
1ØØ LPRINT "Back to normal again"
```

The output produced by this program looks like this:

```
                    Title Line
Report line 2
Report line 3
Report line 4
Condensed print; 132 char/line
Return to normal

This is emphasized print

Back to normal again
```

LSET and RSET Statements

Purpose: Moves data into a random file buffer in preparation for a PUT (file) statement.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: LSET *stringvar* = *x$*

RSET *stringvar* = *x$*

Remarks:

stringvar is the name of a variable defined in a FIELD statement.

x$ is a string expression to place the information into the field identified by *stringvar*.

If *x$* requires fewer bytes than were specified for *stringvar* in the FIELD statement, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If *x$* is longer than *stringvar*, characters are dropped from the right.

Numeric values must be converted to strings before they are LSET or RSET. See "MKI$, MKS$, MKD$ Functions."

See also Appendix A, "BASIC Disk Input and Output," in the *BASIC Handbook* for a complete explanation of using random files.

Note: LSET or RSET can also be used with a string variable that was not defined in a FIELD statement to left-justify or right-justify a string in a given field. For example, the program lines:

```
1Ø A$=SPACE$(2Ø)
2Ø RSET A$=N$
```

right-justify the string N$ in a 2Ø-character field. This can be useful for formatting printed output.

Example: This example converts the numeric value AMT into a string, and left-justifies it in the field A$ in preparation for a PUT (file) statement.

```
1Ø LSET A$=MKS$(AMT)
```

MERGE Command

Purpose: Merges the lines from an ASCII program file into the program currently in memory.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: MERGE *filespec*

Remarks:

filespec is a string expression for the file specification. In BASIC 2.Ø and later releases, it can contain a path. *Filespec* must conform to the rules outlined under "Naming Files" in Chapter 3 of the *BASIC Handbook*; otherwise, an error occurs.

The device is searched for the named file. If found, the program lines in the device file are merged with the lines in memory. If any lines in the file being merged have the same line number as lines in the program in memory, the lines from the file replace the corresponding lines in memory.

After the MERGE command, the merged program resides in memory, and BASIC returns to command level.

In Cassette BASIC, if the device name is omitted, CAS1: is assumed. CAS1: is the only device allowed for MERGE in Cassette BASIC. With Disk BASIC and Advanced BASIC, if the device name is omitted, the DOS default drive is assumed.

If CAS1: is specified as the device name and the filename is omitted, the next ASCII program file encountered on the tape is merged.

If the program being merged was not saved in ASCII format (using the **A** option on the SAVE command), a **Bad file mode** error occurs. The program in memory remains unchanged.

Example: This example merges the file named "NUMBRS" on drive A with the program in memory.

```
MERGE "A:NUMBRS"
```

MID$ Function and Statement

Purpose: Returns the requested part of a given string. When used as a statement, as in the second format, replaces a portion of one string with another string.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: As a function:

*v$* = MID$(*x$*,*n*[,*m*])

As a statement:

MID$(*v$*,*n*[,*m*]) = *y$*

Remarks: **For the function (v$=MID$...):**

x$ is any string expression.

n is an integer expression in the range 1 to 255.

m is an integer expression in the range Ø to 255.

The function returns a string of length *m* characters from *x$* beginning with the *n*th character. If *m* is omitted or if fewer than *m* characters are to the right of the *n*th character, all rightmost characters beginning with the *n*th character are returned. If *m* is equal to Ø, or if *n* is greater than LEN(*x$*), then MID$ returns a null string.

See also "LEFT$" and "RIGHT$" functions.

MID$ Function and Statement

For the statement (MID$...=y$)

v$ is a string variable or array element that will have its characters replaced.

n is an integer expression in the range 1 to 255.

m is an integer expression in the range Ø to 255.

y$ is a string expression.

The characters in *v$*, beginning at position *n*, are replaced by the characters in *y$*. The optional *m* refers to the number of characters from *y$* used in the replacement. If *m* is omitted, all of *y$* is used.

However, regardless of whether *m* is omitted or included, the length of *v$* does not change. For example, if *v$* is four characters long and *y$* is five characters long, then after the replacement *v$* contains only the first four characters of *y$*.

> **Note:** If either *n* or *m* is out of range, an **Illegal function call** error is returned.

MID$ Function and Statement

Example: The first example uses the MID$ function to select the middle portion of the string B$.

```
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
RUN
GOOD EVENING
```

The next example uses the MID$ statement to access substrings imbedded within one large string. This technique reduces fragmentation of string space.

```
10 RECORD$ = STRING$(255,0)
20 PART1.OFF = 1
30 PART1.LEN = 5
40 PART2.OFF = 6
50 PART2.LEN = 15
   .
   .
   .
100 MID$(RECORD$,PART1.OFF,PART1.LEN) = "STRNG"
```

MKDIR Command

Purpose: Creates a directory on the specified disk. (For BASIC 2.Ø and later releases.)

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	

Format: MKDIR *path*

Remarks:

path is a string expression, not exceeding 63 characters, that identifies the new directory to be created. For more information about paths refer to "Naming Files" and "Tree-Structured Directories" in Chapter 3 of the *BASIC Handbook*.

Example: This example creates, from the root directory, a subdirectory called SALES:

```
MKDIR "SALES"
```

This example creates, from the root directory, a subdirectory called MIKE under the directory SALES:

```
MKDIR "SALES\MIKE"
```

This example creates, from the root directory, a subdirectory called ALICE under the directory MIKE:

```
MKDIR "SALES\MIKE\ALICE"
```

This example creates, from the root directory, a subdirectory called ACCTING:

```
MKDIR "ACCTING"
```

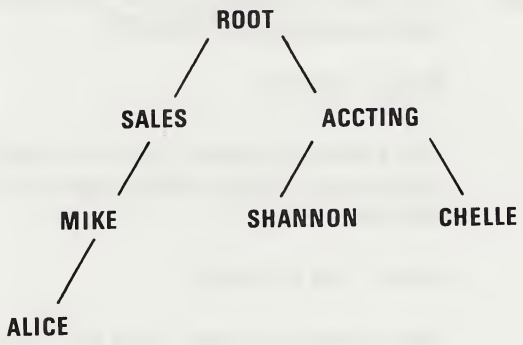
This example makes ACCTING the current directory; then creates two subdirectories called SHANNON and CHELLE:

```
CHDIR "ACCTING"
MKDIR "SHANNON"
MKDIR "CHELLE"
```

The same structure can be created from the root by entering:

```
MKDIR "ACCTING\SHANNON"
MKDIR "ACCTING\CHELLE"
```

By following the above examples, you create a tree structure that looks like this:



MKI$, MKS$, MKD$ Functions

Purpose: Convert numeric type values to string type values.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format:

*v$* = MKI$(*integer expression*)

*v$* = MKS$(*single-precision expression*)

*v$* = MKD$(*double-precision expression*)

Remarks: Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single-precision number to a 4-byte string. MKD$ converts a double-precision number to an 8-byte string.

These functions differ from STR$ because they do not really change the bytes of the data – just the way BASIC interprets those bytes.

See also "CVI, CVS, CVD Functions" in this manual and Appendix A, "BASIC Disk Input and Output," in the *BASIC Handbook*.

MKI$, MKS$, MKD$ Functions

Example: This example uses a random file (#1) with fields defined in line 1ØØ. The first field, D$, is intended to hold a numeric value, AMT. Line 11Ø converts AMT to a string value using MKS$ and uses LSET to place what is really the value of AMT into the random file buffer. Line 12Ø places a string into the buffer (it is not necessary to convert a string); then line 13Ø writes the data from the random file buffer to the file.

```
1ØØ FIELD #1, 4 AS D$, 2Ø AS N$
11Ø LSET D$ = MKS$(AMT)
12Ø LSET N$ = A$
13Ø PUT #1
```

MOTOR Statement

Purpose: Turns the cassette player on and off.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: MOTOR [*state*]

Remarks:

state is a numeric expression indicating on or off.

If *state* is nonzero, the cassette motor is turned on. If *state* is zero, the cassette motor is turned off.

If *state* is omitted, the cassette motor state is switched. That is, if the motor is off, it is turned on and vice versa.

Example: The following sequence of statements turns the cassette motor on, then off, then back on again.

```
10 MOTOR 1
20 MOTOR 0
30 MOTOR
```

NAME Command

Purpose: Changes the name of a disk file. The NAME command in BASIC is similar to the RENAME command in DOS.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: NAME *filespec* AS *filespec*

Remarks:

filespec is a string expression for the file specification. In BASIC 2.Ø and later releases, it can contain a path. *Filespec* must conform to the rules outlined under "Naming Files" in Chapter 3 of the *BASIC Handbook*; otherwise, an error occurs.

filespec is the new filespec. It must be a valid filespec as outlined in the same section.

The file specified by *filespec* must exist and *filename* must not exist on the disk; otherwise, an error results. If the device name is omitted, the DOS default drive is assumed. Note that the file extension does not default to .BAS.

After a NAME command, the file exists on the same disk, in the same disk space, with the new name.

Example: In this example, the file that was formerly named ACCTS.BAS on the disk in drive A is now named LEDGER.BAS.

```
NAME "A:ACCTS.BAS" AS "LEDGER.BAS"
```

NEW Command

Purpose: Deletes the program currently in memory and clears all variables.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: NEW

Remarks: NEW is usually used to remove a program from memory before entering a new program. BASIC always returns to command level after NEW is executed. NEW causes all files to be closed and turns trace off if it was on. See also "TRON and TROFF Commands."

Example: This example deletes the program in memory.

```
NEW
```

OCT$ Function

Purpose: Returns a string that represents the octal value of the decimal argument.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = OCT$(*n*)

Remarks:

n is a numeric expression in the range -32768 to 65535.

If *n* is negative, the twos complement form is used. That is, OCT$(-*n*) is the same as OCT$(65536-*n*).

See "HEX$ Function" for hexadecimal conversion.

Example: This example shows that 24 in decimal is 3Ø in octal.

```
PRINT OCT$(24)
3Ø
```

ON COM(n) Statement

Purpose: Sets up a line number for BASIC to trap to when there is information coming into the communications buffer.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: ON COM(*n*) GOSUB *line*

Remarks:

n is the number of the communications adapter (1 or 2).

line is the line number of the beginning of the trap routine. Setting *line* equal to Ø (zero) disables trapping of communications activity for the specified adapter.

A COM(*n*) ON statement must be executed to activate this statement for adapter *n*. After COM(*n*) ON, if a nonzero line number is specified in the ON COM(*n*) statement, then every time the program starts a new statement, BASIC checks to see if any characters have come in to the specified communications adapter. If so, BASIC performs a GOSUB to the specified *line*.

If COM(*n*) OFF is executed, no trapping takes place for the adapter. Even if communications activity does take place, the event is not remembered.

If a COM(*n*) STOP statement is executed, no trapping takes place for the adapter. However, any

characters being received are remembered so an immediate trap takes place when COM(*n*) ON is executed.

When the trap occurs, an automatic COM(*n*) STOP is executed so that recursive traps never take place. The RETURN from the trap routine automatically does a COM(*n*) ON unless an explicit COM(*n*) OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place, all trapping is automatically disabled (including ON COM, ON ERROR, ON PEN, ON PLAY, ON STRIG, and ON TIMER).

Typically, the communications trap routine reads an entire message from the communications line before returning. It is not recommended that you use the communications trap for single character messages, since at high baud rates the overhead of trapping and reading for each individual character allows the interrupt buffer for communications to overflow.

You can use RETURN *line* if you want to go back to the BASIC program at a fixed line number. This nonlocal return must be used with care, however, since any other GOSUBs, WHILEs, or FORs active at the time of the trap remain active.

Active loops are exited by setting the loop counter variable out of range or setting a conditional statement within the loop to terminate it. This insures that every iteration of a FOR has a corresponding NEXT and every iteration of a WHILE has a corresponding WEND.

ON COM(n) Statement

Example:

```
150 ON COM(1) GOSUB 500
160 COM(1) ON
    .
    .
    .
500     'incoming characters
    .
    .
    .
590 RETURN 300
```

This example sets up a trap routine for the first communications adapter at line 5ØØ.

ON ERROR Statement

Purpose: Enables error trapping and specifies the first line of the error handling subroutine.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: ON ERROR GOTO *line*

Remarks:

line is the line number of the first line of the error trapping routine. If the line number does not exist, an **Undefined line number** error results.

Once error trapping has been enabled, all errors detected (*including direct mode errors*) will cause a jump to the specified error handling subroutine.

To disable error trapping, execute an ON ERROR GOTO Ø. Subsequent errors print an error message and halt execution. An ON ERROR GOTO Ø statement that appears in an error trapping subroutine causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error-trapping subroutines execute an ON ERROR GOTO Ø if an error is encountered for which there is no recovery action.

BASIC considers itself to be within the error trapping routine from the time an error occurs. It branches to the line specified by the ON ERROR statement until a RESUME statement is encountered. You must use the RESUME statement to exit from the error trapping routine. See also "RESUME Statement."

ON ERROR Statement

Because error trapping does not occur within the error trapping routine, an ON ERROR GOTO *line* (within the error trapping routine), where *line* is anything other than Ø, will not work.

> **Note:** If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

Example: This example tests to see if the drive door is open when the program needs to open a file.

```
1Ø ON ERROR GOTO 1ØØ
2Ø OPEN "DATA" FOR INPUT AS #1
3Ø END
   .
   .
   .
1ØØ IF ERR=71 THEN LOCATE 23,1:
    PRINT "DISK IS NOT READY"
11Ø RESUME NEXT
```

ON-GOSUB and ON-GOTO Statements

Purpose: Branches to one of several specified line numbers, depending on the value of an expression.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: ON *n* GOTO *line*[,*line*]...

ON *n* GOSUB *line*[,*line*]...

Remarks:

n is a numeric expression, rounded to an integer, if necessary. It must be in the range Ø to 255, or an **Illegal function call** error occurs.

line is the number of the line to which the program branches.

The value of *n* determines which line number in the list the program uses for branching. For example, if the value of *n* is 3, the third line number in the list is the point at which the program branches.

In the ON–GOSUB statement, each line number in the list must be the first line number of a subroutine. Eventually you must have a RETURN statement to bring you back to the line following the ON–GOSUB.

If the value of *n* is Ø, or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement.

ON-GOSUB and ON-GOTO Statements

Example: The first example branches to line 15Ø if L-1 equals 1; to line 3ØØ if L-1 equals 2; to line 32Ø if L-1 equals 3; and to line 39Ø if L-1 equals 4. If L-1 is equal to Ø, or is greater than 4, then the program just goes on to the next statement.

```
1ØØ ON L-1 GOTO 15Ø,3ØØ,32Ø,39Ø
```

The next example shows how to use an ON–GOSUB statement.

```
1ØØ REM display menu
11Ø PRINT "1. Routine 1"
12Ø PRINT "2. Routine 2"
13Ø PRINT "3. Routine 3"
14Ø PRINT "4. Routine 4"
15Ø INPUT "Your choice?"; CHOICE
16Ø ON CHOICE GOSUB 2ØØ, 3ØØ, 4ØØ,5ØØ
17Ø GOTO 1ØØ ' redisplay menu after routine is done
2ØØ REM start of first routine
    .
    .
    .
29Ø RETURN
3ØØ REM start of second routine
    .
    .
    .
```

ON KEY(n) Statement

Purpose: Sets up a line number for BASIC to trap to when the specified function key or cursor control key is pressed.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: ON KEY(*n*) GOSUB *line*

Remarks:

n is a numeric expression in the range 1 to 2Ø indicating the key to be trapped, as follows:

1-1Ø	Function keys F1 to F1Ø
11	Cursor Up
12	Cursor Left
13	Cursor Right
14	Cursor Down
15-2Ø	keys defined by the form: KEY *n*,CHR$(*KBflag*)+CHR$(*scan code*). (Keys 15-2Ø can be trapped only in Advanced BASIC 2.Ø and later releases.) See "KEY(n) Statement" for more information.

line is the line number of the beginning of the trapping routine for the specified key. Setting *line* equal to Ø stops trapping of the key.

A KEY(*n*) ON statement must be executed to activate this statement. After KEY(*n*) ON, if a nonzero line number is specified in the ON KEY(*n*) statement, then every time the program starts a new

statement, BASIC checks to see if the specified key was pressed. If so, BASIC performs a GOSUB to the specified *line*.

If a KEY(*n*) OFF statement is executed, no trapping takes place for the specified key. Even if the key is pressed, the event is not remembered.

If a KEY(*n*) STOP statement is executed, no trapping takes place for the specified key. However, if the key is pressed the event is remembered, so an immediate trap takes place when KEY(*n*) ON is executed.

When the trap occurs, an automatic KEY(*n*) STOP is executed so that recursive traps never take place. The RETURN from the trap routine automatically does a KEY(*n*) ON unless an explicit KEY(*n*) OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place, all trapping is automatically disabled (including ON COM, ON ERROR, ON PEN, ON PLAY, ON STRIG, and ON TIMER).

Key trapping may not work if you press other keys before the specified key. The key that caused the trap cannot be tested using INPUT$ or INKEY$, so the trap routine for each key must be different if a different function is desired.

You can use RETURN *line* if you want to go back to the BASIC program at a fixed line number. This nonlocal return must be used with care, however, since any other GOSUBs, WHILEs, or FORs active at the time of the trap remain active.

Active loops are exited by setting the loop counter variable out of range or setting a conditional statement within the loop to terminate it. This ensures that every iteration of a FOR has a corresponding NEXT and every iteration of a WHILE has a corresponding WEND.

KEY(*n*) ON has no effect on whether the soft key values are displayed at the bottom of the screen.

Special considerations for DOS national diskettes:

The DOS national diskette keyboard programs have a feature that allows you to change between the United States and national keyboard at any time. Use the F1 or F2 key while holding down Alt and Ctrl to perform the switch. (See *Disk Operating System Reference* for more information on the DOS keyboard programs.) If your BASIC program traps either of these keys, it will not pass the information to the DOS keyboard program and the keyboard change will not take place. If your program needs to provide this ability to change keyboard formats, avoid trapping the F1 and F2 keys.

Note: The shift state you use when trapping either of these keys makes no difference when considering the DOS keyboard programs. Any shift of base state trapping of F1 and F2 prevents the keystroke from being passed to the DOS program.

ON KEY(n) Statement

Example: The following is an example of a trap routine for function key 5.

```
100 ON KEY(5) GOSUB 200
110 KEY(5) ON
   .
   .
   .
200    'function key 5 pressed
   .
   .
   .
290 RETURN 140
```

This example traps Ctrl-Break and Ctrl-Alt-Del. (For BASIC 2.Ø and later releases.)

```
10 KEY 15,CHR$(&H04)+CHR$(70) 'Trap Ctrl-Break
20 KEY 16,CHR$(&H04+&H08)+CHR$(83)
       'Trap Ctrl-Alt-Del
30 ON KEY(15) GOSUB 1000
40 ON KEY(16) GOSUB 2000
50 KEY(15) ON: KEY(16) ON
   .
   .
   .
1000 PRINT "Trapping for Ctrl-Break"
1010 RETURN
2000 TRAPS=TRAPS+1
2010 ON TRAPS GOTO 2100,2200,2300,2400, 2500
2020 '
2100 PRINT "First trap of System Reset":RETURN
2200 PRINT "Second trap of System Reset":RETURN
2300 PRINT "Third trap of System Reset":RETURN
2400 PRINT "Fourth trap of System Reset":RETURN
2500 KEY(16) OFF 'Disable trap of System Reset
```

ON PEN Statement

Purpose: Sets up a line number for BASIC to transfer control to when the light pen is activated.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: ON PEN GOSUB *line*

Remarks:

line is the line number of the beginning of the trap routine for the light pen. Using a line number of Ø disables trapping of the light pen.

A PEN ON statement must be executed to activate this statement. After PEN ON, if a nonzero line number is specified in the ON PEN statement, then every time the program starts a new statement BASIC checks to see if the pen was activated. If so, BASIC performs a GOSUB *line*.

If PEN OFF is executed, no trapping takes place. Even if the light pen is activated, the event is not remembered.

If a PEN STOP statement is executed, no trapping takes place, but pen activity is remembered so that an immediate trap takes place when PEN ON is executed.

When the trap occurs, an automatic PEN STOP is executed so recursive traps never take place. The RETURN from the trap routine automatically does a PEN ON unless an explicit PEN OFF was performed inside the trap routine.

ON PEN Statement

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ON COM, ON ERROR, ON PEN, ON PLAY, ON STRIG, and ON TIMER).

PEN(Ø) is not set when pen activity causes a trap.

You can use RETURN *line* if you want to go back to the BASIC program at a fixed line number. This nonlocal return must be used with care, however, since any other GOSUBs, WHILEs, or FORs active at the time of the trap remain active.

Active loops are exited by setting the loop counter variable out of range or setting a conditional statement within the loop to terminate it. This insures that every iteration of a FOR has a corresponding NEXT and every iteration of a WHILE has a corresponding WEND.

Note: Do not perform cassette I/O while PEN is ON.

Example: This example sets up a trap routine for the light pen.

```
1Ø ON PEN GOSUB 5ØØ
2Ø PEN ON
   .
   .
   .
5ØØ 'subroutine for pen
   .
   .
   .
65Ø RETURN 3Ø
```

ON PLAY(n) Statement

Purpose: Plays continuous background music during program execution. (For BASIC 2.Ø and later releases.)

Versions: Cassette Disk Advanced Compiler
***

Format: ON PLAY(*n*) GOSUB *line*

Remarks:

n is an integer expression in the range 1 to 32 indicating the notes to be trapped. Values entered outside this range result in an **Illegal function call** error.

line is the beginning line number of the trap routine for PLAY. A line number of Ø stops the trapping of PLAY.

A PLAY ON statement must be used to start the ON PLAY(n) statement. After PLAY ON, if a nonzero line number is specified in the PLAY(*n*) statement, each time the program starts a new statement BASIC checks to see if the music buffer has gone from *n* to *n*-1 notes. If so, BASIC performs a GOSUB to the specified line.

If PLAY OFF is used, no trapping takes place. Even if a play activity takes place, the event is not remembered.

If a PLAY STOP statement is used, no trapping takes place, but play activity is remembered so that an immediate trap takes place when PLAY ON is executed.

When the trap occurs, an automatic PLAY STOP is run so recursive traps never take place. The RETURN from the trap routine automatically does a PLAY ON unless an explicit PLAY OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not running a program. When an error trap (resulting from an ON ERROR statement) takes place, all trapping is automatically disabled (including ON COM, ON ERROR, ON PEN, ON PLAY, ON STRIG, and ON TIMER).

You can use RETURN *line* if you want to go back to the BASIC program at a fixed line number. This nonlocal return must be used with care, however, since any other GOSUBs, WHILEs, or FORs active at the time of the trap remain active.

Active loops are exited by setting the loop counter variable out of range or setting a conditional statement within the loop to terminate it.

Notes:

1. A PLAY event trap is issued only when PLAY is in the Music Background mode (PLAY "MB..."). An event trap is not issued when PLAY is in the Music Foreground mode (PLAY "MF...").

2. A PLAY event trap is not issued if the Music Background buffer is already empty when a PLAY ON statement is performed.

3. Be careful choosing values for *n*. For example: ON PLAY(32) causes so many event traps that little time remains to run the rest of the program.

ON PLAY(n) Statement

See also "PLAY(n) Function" in this chapter for additional information.

Example: This example sets up a trap routine that is invoked when five notes are left in the background music buffer.

```
10 ON PLAY(5) GOSUB 500
20 PLAY ON
   .
   .
   .
500    'subroutine for background music
   .
   .
   .
650 RETURN 30
```

ON STRIG(n) Statement

Purpose: Sets up a line number for BASIC to trap to when one of the joystick buttons (triggers) is pressed.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: ON STRIG(*n*) GOSUB *line*

Remarks:

n can be Ø, 2, 4, or 6, and indicates the button to be trapped as follows:

Ø button A1

2 button B1

4 button A2

6 button B2

line is the line number of the beginning of the trap routine for STRIG. A line number of Ø stops trapping of the joystick button.

A STRIG(*n*) ON statement must be executed to activate this statement for button *n*. If STRIG(*n*) ON is executed and a nonzero line number is specified in the ON STRIG(*n*) statement, then every time the program starts a new statement BASIC checks to see if the specified button has been pressed. If so, BASIC performs a GOSUB to the specified *line*.

If STRIG(*n*) OFF is executed, no trapping takes place for button *n*. Even if the button is pressed, the event is not remembered.

If a STRIG(*n*) STOP statement is executed, no trapping takes place for button *n*, but the button being pressed is remembered so that an immediate trap takes place when STRIG(*n*) ON is executed.

When the trap occurs, an automatic STRIG(*n*) STOP is executed so that recursive traps never take place. The RETURN from the trap routine automatically does a STRIG(*n*) ON unless an explicit STRIG(*n*) OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place, all trapping is automatically disabled (including ON COM, ON ERROR, ON PEN, ON PLAY, ON STRIG, and ON TIMER).

Using STRIG(*n*) ON activates the interrupt routine that checks the button status for the specified joystick button. Downstrokes that cause trapping do not set functions STRIG(Ø), STRIG(2), STRIG(4), or STRIG(6).

You can use RETURN *line* to go back to the BASIC program at a fixed line number. This nonlocal return must be used with care, however, since any other GOSUBs, WHILEs, or FORs active at the time of the trap remain active.

Active loops are exited by setting the loop counter variable out of range or setting a conditional statement within the loop to terminate it. This insures that every iteration of a FOR has a corresponding NEXT and every iteration of a WHILE has a corresponding WEND.

ON STRIG(n) Statement

Example: This is an example of a trapping routine for the button on the first joystick.

```
10 ON STRIG(0) GOSUB 2000
20 STRIG(0) ON
    .
    .
    .
500 'subroutine for 1st button
    .
    .
    .
650 RETURN
```

ON TIMER Statement

Purpose: Transfers control to a given line number in a BASIC program when a defined period of time has elapsed. (For BASIC 2.Ø and later releases.)

Versions:

Cassette	Disk	Advanced	Compiler
		***	

Format: ON TIMER(*n*) GOSUB *line*

Remarks:

n is a numeric expression in the range 1 to 86,4ØØ (1 second through 24 hours). Values entered that are outside this range result in an **Illegal function call** error.

line is the beginning line number of the trap routine for TIMER. A line number of Ø stops timer trapping.

A TIMER ON statement must be used to start the ON TIMER statement. After TIMER ON, specifying a nonzero line number in the ON TIMER(*n*) statement causes BASIC to keep track of the passing seconds. When *n* seconds have elapsed, BASIC performs a GOSUB to the specified line. The event trap occurs, and BASIC starts counting again from Ø.

If TIMER OFF is used, no trapping takes place. Even if TIMER activity takes place, the event is not remembered.

If a TIMER STOP statement is used, no trapping takes place, but TIMER activity is remembered so that an immediate trap occurs when TIMER ON is used.

When the trap occurs, an automatic TIMER STOP is executed so that recursive traps never take place. The RETURN from the trap routine automatically does a TIMER ON unless an explicit TIMER OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not running a program. When an error trap (resulting from an ON ERROR statement) takes place, all trapping is automatically disabled (including ON COM, ON ERROR, ON PEN, ON PLAY, ON STRIG, and ON TIMER).

You can use RETURN *line* to go back to the BASIC program at a fixed line number. This nonlocal return must be used with care, however, since any other GOSUBs, WHILEs, or FORs active at the time of the trap remain active.

Active loops are exited by setting the loop counter variable out of range or setting a conditional statement within the loop to terminate it. This insures that every iteration of a FOR has a corresponding NEXT and every iteration of a WHILE has a corresponding WEND.

Example: ON TIMER is useful in programs that need an interval timer. This example displays the time of day on line 1 every minute.

```
10 CLS
20 ON TIMER(60) GOSUB 10000
30 TIMER ON
   .
   .
   .
10000 OLDROW=CSRLIN 'save current row
10010 OLDCOL=POS(0) 'save current column
10020 LOCATE 1,1:PRINT TIME$;
10030 LOCATE OLDROW,OLDCOL 'restore row & col
10040 RETURN
```

OPEN Statement

Purpose: Allows input or output to a file or device.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: First form:

OPEN *filespec* [FOR *mode*] AS [#]*filenum* [LEN=*recl*]

Alternate form:

OPEN *mode2*, [#]*filenum, filespec* [,*recl*]

Remarks:

mode (first form) is one of the following:

OUTPUT specifies sequential output mode.

INPUT specifies sequential input mode.

APPEND specifies sequential output mode where the file is positioned to the end of data on the file when it is opened.

Note that *mode* must be a string constant, *not* enclosed in quotation marks. If *mode* is omitted, random access is assumed.

mode2 (alternate form) is a string expression with the first character being one of the following:

O specifies sequential output mode

I specifies sequential input mode

R specifies random input/output mode

For both formats:

filenum is an integer expression whose value is between 1 and the maximum number of files allowed. In Cassette BASIC, the maximum number is 4. In Disk BASIC and Advanced BASIC, the default maximum is 3, but this can be changed with the **/F:** switch on the BASIC command line.

filespec is a string expression for the file specification. In BASIC 2.Ø and later releases, it can contain a path. *Filespec* must conform to the rules outlined under "Naming Files" in Chapter 3 of the *BASIC Handbook*; otherwise, an error occurs.

recl is an integer expression which, if included, sets the record length for random files. It can range from 1 to 32767. The default record length is 128 bytes. *recl* cannot exceed the value set by the **/S:** switch in the BASIC command line.

OPEN allocates a buffer for I/O to the file or device and determines the mode of access that is used with the buffer.

filenum is the number that is associated with the file or device for as long as it is open and is used by other I/O statements to refer to the file or device.

OPEN Statement

An OPEN must be executed before any I/O can be done to a device or file using any of the following statements, or any statement or function requiring a file number:

PRINT #	WRITE #
PRINT # USING	INPUT$
INPUT #	GET #
LINE INPUT #	PUT #
IOCTL #	

GET and PUT are valid for random files or communications files. A disk file cay be either random or sequential, and a printer can be opened in either random or sequential mode; however, all other standard devices can be opened only for sequential operations. See "OPEN "COM... Statement."

BASIC normally adds a line feed after each carriage return (CHR$(13)) sent to a printer. However, if you open a printer (LPT1:,LPT2:, or LPT3:) as a random file with width 255, this line feed is suppressed.

APPEND is valid only for disk files. The file pointer is initially set to the end of the file, and the record number is set to the last record of the file. PRINT # or WRITE # then extends the file.

A file cannot be opened for sequential output or append if the file is already open.

If the device name is omitted when you are using Cassette BASIC, CAS1: is assumed. If you are using Disk BASIC or Advanced BASIC, the DOS default drive is assumed.

If CAS1: is specified as the device and the filename is omitted, then the next data file on the cassette is opened.

In Cassette BASIC, a maximum of four files can be open at one time (cassette, printer, keyboard, and screen). Note that only one cassette file can be open at a time. For Disk BASIC and Advanced BASIC the default maximum is three files. You can override this value by using the **/F:** option in the BASIC command line.

If a file opened for input does not exist, a **File not found** error occurs. If a file that does not exist is opened for output, append, or random access, a file is created.

Any values given outside the ranges indicated result in an **Illegal function call** error. The file is not opened.

See also the section on device drivers in Chapter 3 of the *BASIC Handbook*, as well as Appendix A of the *BASIC Handbook*, for a complete explanation of using disk files. See also "OPEN "COM Statement" for information on opening communications files.

Examples: Either of the following statements opens the file named "DATA" for sequential output on the default device (CAS1: for Cassette BASIC, default drive for Disk BASIC and Advanced BASIC).

```
1Ø OPEN "DATA" FOR OUTPUT AS #1
  or
1Ø OPEN "O",#1,"DATA"
```

In the preceding example, note that opening for output destroys any existing data in the file. If you do not wish to destroy data, you must open for APPEND.

Either of the following two statements opens the file named "SSFILE" on the disk in drive B for random input and output. The record length is 256.

```
1Ø OPEN "B:SSFILE" AS 1 LEN=256
   or
1Ø OPEN "R",1,"B:SSFILE",256
```

This example opens the file "DATA.ART" on the disk in drive A and positions the file pointers so that any output to the file is placed at the end of existing data in the file.

```
10 FILE$ = "A:DATA.ART"
2Ø OPEN FILE$ FOR APPEND AS 3
```

Line 1Ø in the next example opens the printer in random mode. Because the default width is 8Ø, the lines printed by lines 2Ø and 3Ø end with a carriage return/line feed. Line 4Ø changes the printer width to 255, so the line feed after the carriage return is suppressed. Therefore, the line printed by line 5Ø ends only with a carriage return and not a line feed. This causes the line printed by line 7Ø to overprint "This line will be underlined", causing the line to be underlined. Line 6Ø changes the width back to 8Ø so the underlines and following lines end with a line feed.

```
10 OPEN "LPT1:" AS #1
20 PRINT #1,"Printing width 80"
30 PRINT #1,"Now change to width 255"
40 WIDTH #1,255
50 PRINT #1,"This line will be underlined"
60 WIDTH #1,80
70 PRINT #1, STRING$(28,"_")
80 PRINT #1,"Printing width 80 with CR/LF"
RUN

Printing width 80
Now change to width 255
This line will be underlined
Printing width 80 with CR/LF
```

The following examples illustrate the use of paths for filespec.

Either of these statements opens the file called "DATA" for sequential output on the default device in the directory called LVL2.

```
10 OPEN "LVL1\LVL2\DATA" FOR OUTPUT AS #1
  or
10 OPEN "O",#1,"LVL1\LVL2\DATA"
```

Either of the next two statements opens the file named "RRFILE" in the LVL1 directory on the disk in drive B for random input and output. The record length is 256.

```
20 OPEN "B:LVL1\RRFILE" AS 1 LEN=256
or
20 OPEN "R",B:LVL1\RRFILE:",256
```

OPEN "COM...
Statement

Purpose: Opens a communications file.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	(**)

Valid only with Asynchronous Communications Adapter.

Format: OPEN "COM*n*:[*speed*] [,*parity*] [,*data*] [,*stop*] [,RS] [,CS[*n*]] [,DS[*n*]] [,CD[*n*]] [,LF] [,PE]" AS [#]*filenum* [LEN=*number*]

Remarks:

n is 1 or 2, indicating the number of the Asynchronous Communications Adapter.

speed is an integer constant specifying the transmit/receive bit rate in bits per second (bps). Valid speeds are 75, 11Ø, 15Ø, 3ØØ, 6ØØ, 12ØØ, 18ØØ, 24ØØ, 48ØØ, and 96ØØ. The default is 3ØØ bps.

parity is a one-character constant specifying the parity for transmit and receive as follows:

S SPACE: Parity bit always transmitted and received as a space (Ø bit).

O ODD: Odd transmit parity; odd receive parity checking.

M MARK: Parity bit always transmitted and received as a mark (1 bit).

E EVEN: Even transmit parity; even receive parity checking.

N NONE: No transmit parity; no receive parity checking.

The default is EVEN (E).

data is an integer constant indicating the number of transmit/receive data bits. Valid values are: 5, 6, 7, or 8. The default is 7.

stop is an integer constant indicating the number of stop bits. Valid values are 1 or 2. The default is two stop bits for 75 and 11Ø bps; one stop bit for all others. If you use 5 for *data*, a 2 here means 1-1/2 stop bits.

filenum is an integer expression that evaluates to a valid file number. The number is then associated with the file for as long as it is open and is used by other communications I/O statements to refer to the file.

number is the maximum number of bytes that can be read from the communication buffer when using GET or PUT. The default is 128 bytes.

OPEN "COM.. allocates a buffer for I/O in the same fashion as OPEN for disk files. It supports RS232 asynchronous communication with other computers and peripherals.

A communications device can be open to only one file number at a time.

The **RS, CS, DS, CD, LF** and **PE** options affect the line signals as follows:

RS	suppresses RTS (Request To Send)
CS[n]	controls CTS (Clear To Send)
DS[n]	controls DSR (Data Set Ready)
CD[n]	controls CD (Carrier Detect)
LF	sends a line feed following each carriage return
PE	enables parity checking

The CD (Carrier Detect) is also known as the RLSD (Received Line Signal Detect).

> **Note:** The *speed, parity, data*, and *stop* parameters are positional, but **RS, CS, DS, CD, LF**, and **PE** are not.

The RTS (Request To Send) line is turned on when you execute an OPEN "COM.. statement unless you include the **RS** option.

The *n* argument in the **CS, DS**, and **CD** options specifies the number of milliseconds to wait for the signal before returning a **Device timeout** error. *n* can range from Ø to 65535. If *n* is omitted or is equal to zero, then the line status is not checked at all.

The defaults are **CS1ØØØ, DS1ØØØ**, and **CDØ**. If **RS** was specified, **CSØ** is the default.

That is, normally I/O statements to a communications file fail if the CTS (Clear To Send) or DSR (Data Set Ready) signals are off. The

system waits 1 second before returning a **Device timeout**. The **CS** and **DS** options allow you to ignore these lines or to specify the amount of time to wait before the timeout.

Normally Carrier Detect (CD or RLSD) is ignored when an OPEN "COM... statement is executed. The **CD** option allows you to test this line by including the *n* parameter, in the same way as **CS** and **DS**. If *n* is omitted or is equal to zero, then Carrier Detect is not checked at all (which is the same as omitting the **CD** option).

The **LF** parameter is intended for those using communications files to print to a serial line printer. When you specify **LF**, a line feed character (hexØA) is automatically sent after each carriage return character (hex ØC). (This includes the carriage return sent as a result of the width setting.) INPUT # and LINE INPUT #, when used to read from a communications file that was opened with the **LF** option, stop when they see a carriage return. The line feed is always ignored.

The **PE** option enables parity checking. The default is no parity checking. The **PE** option causes a **Device I/O error** on parity errors and turns on the high order bit for 7 or less data bits. The **PE** option does *not* affect framing and overrun errors. These errors always turn on the high order bit and cause a **Device I/O error**.

Any coding errors within the string expression starting with *speed* result in a **Bad file name** error. No indication is given as to which parameter is in error.

See also Appendix C, "Communications," for more information on control of output signals, as well as other technical information on communications support.

If you specify 8 data bits, you must specify parity N. BASIC uses all 8 bits in a byte to store numbers, so if you are transmitting or receiving numeric data (for example, by using PUT), you must specify 8 data bits. (This is not necessary if you are sending numeric data *as text*.)

See also the previous entry, "OPEN Statement," for information on opening devices other than communications devices.

Example: In this example, file 1 is opened for communication with all defaults. The speed is 3ØØ bps with even parity. There are 7 data bits and 1 stop bit.

```
1Ø OPEN "COM1:" AS #1
```

In this example, file 2 is opened for asynchronous I/O at 12ØØ; no parity is to be produced or checked; 8-bit bytes are sent and received; and 1 stop bit is transmitted.

```
1Ø OPEN "COM2:12ØØ,N,8" AS #1
```

This example opens COM1: at 96ØØ bps with no parity and 8 data bits. CTS, DSR, and CD are not checked.

```
1Ø OPEN "COM1:96ØØ,N,8,,CS,DS,CD" AS #1
```

This example opens COM1: at 12ØØ bps with the defaults of even parity, 7 data bits, and 1 stop bit. RTS is sent, CTS is not checked, and **Device timeout** is given if DSR is not seen within 2 seconds. The

commas are required to indicate the position of the *parity, start*, and *stop* parameters, even though a value is not specified. This is what is meant by *positional* parameters.

```
10 OPEN "COM1:1200,,,,CS,DS2000" AS #1
```

An OPEN statement can be used with an ON ERROR statement to make sure a modem is working properly before sending any data. For example, the following program makes sure you get Carrier Detect (CD or RLSD) from the modem before starting. Line 20 | is set to timeout after 10 seconds. TRIES is set to 6 so we give up if Carrier Detect is not seen within 1 minute. Once communication is established, the file reopens with a shorter delay until timeout.

```
10 TRIES=6:ON ERROR GOTO 100
20 OPEN "COM1:300,N,8,2,CS,DS,CD10000" AS #1
30 ON ERROR GOTO 0
40 CLOSE #1 'Works so can continue
50 GOTO 1000
   .
   .
   .
100 TRIES=TRIES-1
110 IF TRIES=0 THEN ON ERROR GOTO 0  'give up
120 RESUME
   .
   .
   .
1000 OPEN "COM1:300,N,8,2,CS,DS,CD2000" AS #1
```

OPEN "COM...
Statement

The next example shows a typical way to use a communication file to control a serial line printer. The **LF** parameter in the OPEN statement ensures that lines do not print on top of each other.

```
10 WIDTH "COM1:", 132
20 OPEN "COM1:1200,N,8,,CS10000,
   DS10000,CD10000,LF" AS #1
```

OPTION BASE Statement

Purpose: Declares the minimum value for array subscripts.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: OPTION BASE *n*

Remarks: *n* is 1 or Ø.

The default base is Ø. If the statement:

```
OPTION BASE 1
```

is executed, the lowest value an array subscript can have is 1.

The OPTION BASE statement must be coded *before* you define or use any arrays. An error occurs if you change the base value when arrays exist.

In BASIC 1.1Ø, the program you are chaining to cannot have an OPTION BASE statement even when both programs have the same base value.

OUT
Statement

Purpose: Sends a byte to a machine output port.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: OUT *n,m*

Remarks:

n is a numeric expression for the port number, in the range Ø to 65535.

m is a numeric expression for the data to be transmitted, in the range Ø to 255.

See also the IBM Personal Computer *Technical Reference* manual for a description of valid port numbers (I/O addresses).

OUT is the complementary statement to the INP function. See also "INP Function."

One use of OUT is to affect the video output. On some displays attached to the Color/Graphics Monitor Adapter, you may find that the first two or three characters on the line don't show up on the screen. If your display does not have a horizontal adjustment control, you can use the following statements to shift the display:

```
OUT 98Ø,2: OUT 981,43
```

This shifts the display two characters to the right in 4Ø-column width (or 16 points in medium resolution graphics mode, or 32 points in high resolution graphics mode).

```
OUT 98Ø,2: OUT 981,85
```

This shifts the display five characters to the right in 8Ø-column width.

The shift caused by these OUT statements remains in effect until a WIDTH or SCREEN statement is executed. The MODE command from DOS can also be used to shift the display as described here; it has the benefit of remaining in effect until a System Reset.

Example: This sends the value 1ØØ to output port 32.

```
1ØØ OUT 32,1ØØ
```

PAINT
Statement

Purpose: Fills in an area on the screen with the selected color.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Graphics mode only.

Format: PAINT (*x,y*) [[,*paint*] [,*boundary*] [,*background*]]

Remarks:

(x,y) are the coordinates of a point within the area to be filled in. The coordinates can be given in absolute or relative form (see "Specifying Coordinates" under "Graphics Modes" in Chapter 3 of the *BASIC Handbook*). This point is used as a starting point. Points specified outside the limits of the screen are not plotted, and no error occurs.

paint can be a numeric or string expression. It is used to fill a color or pattern in or around a bounded area. When *paint* is a numeric expression, it chooses an attribute from the legal attribute range for the current screen mode. In medium resolution, the color is the current color for that attribute defined by the COLOR statement. Four attributes (Ø-3) are available in medium resolution. In high resolution, two attributes (Ø-1) are available. Zero is always the attribute for the background. The default foreground attribute is the

maximum attribute for that screen mode: 3 in medium resolution; 1 in high resolution.

If *paint* is a string expression, then "tiling" is performed, as described later in this entry. Paint tiling is valid for BASIC 2.Ø and later releases.

boundary is a numeric expression that evaluates to an integer in the legal attribute range of the current screen mode. It defines the attribute for the edges of the figure to be painted.

background
is a 1-byte string expression used in paint tiling. (For BASIC 2.Ø and later releases.)

Since there are only two attributes in high resolution, *paint* should not be different from *boundary*. By default, *boundary* is equal to *paint*. You can paint either a white area black or a black area white.

In medium resolution, you can fill inside or around a defined area with any one of four colors from the current palette defined by the COLOR statement. An example of this is filling a red circle with green, or surrounding a red circle with green.

paint begins at the specified starting point and covers an area until it meets the specified boundary attribute. Therefore, *paint* must always begin inside the area to be painted. If the specified starting point already has the same attribute as *boundary*, then painting stops at that point and appears not to occur. An example of this is plotting a point with PSET that

has the same attribute as *boundary*, and then using the coordinates of that point with the PAINT statement.

PAINT fills any designated area no matter what the shape of the area; however, the more complex the edges of a figure (jagged edges, for instance), the more stack space BASIC uses. Under these circumstances you may want to use the CLEAR statement at the beginning of your program to increase the stack space.

The PAINT statement allows scenes to be displayed with very few statements.

In the example that follows, the PAINT statement in line 3Ø fills in the box drawn in line 2Ø with the color represented by the attribute in the current palette.

```
1Ø SCREEN 1
2Ø LINE (Ø,Ø)-(1ØØ,15Ø),2,B
3Ø PAINT (5Ø,5Ø),1,2
```

The following discussion deals with paint tiling only. (For BASIC 2.Ø and later releases.)

To use paint tiling, the *paint* attribute must be a string expression in the form:

```
CHR$(&Hnn)+CHR$(&Hnn)+CHR$(&Hnn)
```

The CHR$ sequence specifies a bit mask that is 1 byte wide. When the mask is plotted all the way across and down the designated area defined by *boundary*, a pattern is created rather than a solid color. You design the pattern. The two hexadecimal numbers in the CHR$ expression correspond to 8

bits, or 1 byte. The string expression can contain up to 64 bytes. The design created by the string expression can be mapped as follows:

```
        x increases -->
        7 6 5 4 3 2 1Ø
Ø,Ø     x x x x x x x x      Tile byte Ø
Ø,1     x x x x x x x x      Tile byte 1
Ø,2     x x x x x x x x      Tile byte 2
.
.
.
Ø,63    x x x x x x x x      Tile byte 63
                             (maximum allowed)
```

The tile pattern is repeated uniformly over the area defined by *boundary*. If you do not define an area, the whole screen is your designated area. Each byte of the tile string masks 8 bits along the **x** axis when plotting points. Each byte of the tile string is rotated as required to align the pattern along the **y** axis. BASIC chooses the particular byte of the pattern to plot, using the formula **y mod tile length**.

PAINT
Statement

Because there is only 1 bit per pixel in high resolution, a point is plotted at every position in the bit mask that has a value of 1. In high resolution, the screen can be painted with x's using the following example:

```
10 CLS:SCREEN 1: COLOR 1: KEY OFF
20 LOCATE 12,7:PRINT "I JUST LOVE MY IBM COMPUTER"
30 PAINT(320,100),CHR$(&H81)+CHR$(&H42)+
   CHR$(&H24)+CHR$(&H18)+CHR$(&H24)+
   CHR$(&H24)+CHR$(&H42)+CHR$(&H81)
40 GOTO 40
```

The length of this mask is 8, indexed Ø through 7. In this case, PAINT at coordinates (32Ø,1ØØ) begins by plotting byte 4. This is calculated using the **y mod tile length** formula by substituting 1ØØ for **y** and 8 for **tile length**. This pattern appears on the screen as:

	x increases -->	
Tile byte Ø	1 Ø Ø Ø Ø Ø Ø 1	CHR$(&H81)
Tile byte 1	Ø 1 Ø Ø Ø Ø 1 Ø	CHR$(&H42)
Tile byte 2	Ø Ø 1 Ø Ø 1 Ø Ø	CHR$(&H24)
Tile byte 3	Ø Ø Ø 1 1 Ø Ø Ø	CHR$(&H18)
Tile byte 4	Ø Ø Ø 1 1 Ø Ø Ø	CHR$(&H18)
Tile byte 5	Ø Ø 1 Ø Ø 1 Ø Ø	CHR$(&H24)
Tile byte 6	Ø 1 Ø Ø Ø Ø 1 Ø	CHR$(&H42)
Tile byte 7	1 Ø Ø Ø Ø Ø Ø 1	CHR$(&H81)

The method of designing patterns in each screen varies depending on the number of color attributes available in each screen mode. This is so because the number of bits per pixel is directly related to the number of color attributes available in each screen mode. In any screen, where X is the total number of color attributes for that screen,

$$LOG_2(X)=Y$$

where Y is the number of bits per pixel. In high resolution, each byte of the string is able to plot 8 points across the screen (1 bit per pixel), since $LOG_2(2)=1$.

In Screen 1, one medium-resolution tile byte describes 4 pixels, since medium resolution has only 2 bits per pixel: that is, $LOG_2(4)=2$ bits per pixel. Every 2 bits of the tile byte describes 1 of 4 possible color attributes associated with each of the 4 pixels to be plotted.

The following chart shows the binary and hexadecimal values associated with each attribute in medium resolution.

Color palette Ø	Attrib. in binary	Pattern to draw solid line in binary	Pattern to draw solid line in hexadecimal
green	Ø1	Ø1Ø1Ø1Ø1	&H55
red	1Ø	1Ø1Ø1Ø1Ø	&HAA
brown	11	11111111	&HFF

Color palette 1	Attrib. in binary	Pattern to draw solid line in binary	Pattern to draw solid line in hexadecimal
cyan	Ø1	Ø1Ø1Ø1Ø1	&H55
magenta	1Ø	1Ø1Ø1Ø1Ø	&HAA
white	11	11111111	&HFF

In medium resolution, the following example plots a pattern of boxes with a border color of red in palette Ø and magenta in palette 1.

```
1Ø CLS: SCREEN 1: COLOR 1: KEY OFF
2Ø LOCATE 12,7:PRINT "I JUST LOVE MY IBM COMPUTER"
3Ø PAINT (32Ø,1ØØ),CHR$(&HAA)+CHR$(&H82)+
   CHR$(&H82)+CHR$(&H82)+CHR$(&H82)+
   CHR$(&H82)+CHR$(&H82)+CHR$(&HAA)
4Ø GOTO 4Ø
```

	7 6 5 4 3 2 1 Ø	
Tile byte Ø	1 Ø 1 Ø 1 Ø 1 Ø	CHR$(&HAA)
Tile byte 1	1 Ø Ø Ø Ø Ø 1 Ø	CHR$(&H82)
Tile byte 2	1 Ø Ø Ø Ø Ø 1 Ø	CHR$(&H82)
Tile byte 3	1 Ø Ø Ø Ø Ø 1 Ø	CHR$(&H82)
Tile byte 4	1 Ø Ø Ø Ø Ø 1 Ø	CHR$(&H82)
Tile byte 5	1 Ø Ø Ø Ø Ø 1 Ø	CHR$(&H82)
Tile byte 6	1 Ø Ø Ø Ø Ø 1 Ø	CHR$(&H82)
Tile byte 7	1 Ø 1 Ø 1 Ø 1 Ø	CHR$(&HAA)

Occasionally, you may want to tile over an already painted area that is the same color or pattern as two consecutive bytes in the tile mask. Normally, this constitutes a terminating condition because your point is surrounded by points of the same bit pattern. (An example follows on the use of *background*.)

You can use the *background* attribute to skip this terminating condition. You cannot specify more than two consecutive lines in the tile pattern that matches this *background* attribute. Doing so causes an **Illegal function call** error.

PAINT
Statement

Example: The program below demonstrates how to tile an area with three lines of red, two lines of green, and one line of red. The palette then changes to show that the same tile mask yields the same pattern with different colors.

```
10 CLS:SCREEN 1,0:KEY OFF
20 TIL$=CHR$(&HAA)+CHR$(&HAA)+CHR$(&HAA)
   +CHR$(&H55)+CHR$(&H55)+CHR$(&HFF)
30 COLOR 0,0  'choose palette 0
40 VIEW (1,1)-(150,100),0,2
50 GOSUB 1000
60 COLOR 0,1  'choose palette 1
70 GOTO 1020
1000 PAINT (125,50),TIL$,2
1010 RETURN
1020 GOTO 1020
```

The following example uses paint tiling with the *background* attribute.

```
10 CLS:SCREEN 1:COLOR 0,1:KEY OFF
20 TIL$=CHR$(&H5F)+CHR$(&H5F)+CHR$(&H27)+CHR$(&H81)
30 VIEW (1,1)-(150,100),0,2
40 LOCATE 3,22:PRINT "<---Without back-"
50 LOCATE 4,22:PRINT " ground tile "
60 PAINT (125,50),CHR$(&H5F)
70 PAINT (125,50),TIL$,2
80 '
90 'with background tile'
100 '
110 VIEW (160,100)-(310,198),0,2
120 LOCATE 16,1:PRINT "With background-->"
130 LOCATE 17,1:PRINT "tile chr$(&H5F)"
140 PAINT (125,50),CHR$(&H5F)
150 PAINT (125,50),TIL$,2,CHR$(&H5F)
160 LINE (1,100)-(319,100),3
170 FOR I=1 TO 2500:NEXT I
180 GOTO 180
```

PEEK
Function

Purpose: Returns the byte read from the indicated memory position.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = PEEK(*n*)

Remarks:

n is an integer in the range Ø to 65535. *n* is the offset from the current segment as defined by the DEF SEG statement. See "DEF SEG Statement."

The returned value is an integer in the range Ø to 255.

PEEK is the complementary function to the POKE statement. See "POKE Statement."

Example: The following example in a program tests which display adapter is on the system. After line 3Ø is executed, the variable IBMMONO has a value of Ø if the IBM Color/Graphics Monitor Adapter is used, or 1 if the IBM Monochrome Display and Parallel Printer Adapter is used.

```
1Ø 'test display adapter
2Ø DEF SEG=Ø
3Ø IF (PEEK(&H41Ø) AND &H3Ø)=&H3Ø
      THEN IBMMONO=1
      ELSE IBMMONO=Ø
```

PEN
Statement and Function

Purpose: Reads the light pen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

PEN STOP only in Advanced BASIC and Compiler.

Format: As a statement:

PEN ON

PEN OFF

PEN STOP

As a function:

v = PEN(n)

Remarks: The PEN function, v=PEN(n), reads the light pen coordinates.

n is a numeric expression in the range Ø to 9, and affects the value returned by the function as follows:

Ø A flag indicating if pen was down since last poll. Returns -1 if down, Ø if not.

1 Returns the *x* coordinate where pen was last activated. Range is Ø to 319 in medium resolution, and Ø to 639 in high resolution.

PEN
Statement and Function

2 Returns the *y* coordinate where pen was last activated. Range is Ø to 199.

3 Returns the current pen switch value. -1 if down, Ø if up.

4 Returns the last known valid *x* coordinate. Range is Ø to 319 in medium resolution, and Ø to 639 in high resolution.

5 Returns the last known valid *y* coordinate. Range is Ø to 199.

6 Returns the character row position where pen was last activated. Range is 1 to 24.

7 Returns the character column position where pen was last activated. Range is 1 to 4Ø or 1 to 8Ø, depending on WIDTH.

8 Returns the last known valid character row. Range is 1 to 24.

9 Returns the last known valid character column position. Range is 1 to 4Ø or 1 to 8Ø, depending on WIDTH.

PEN ON enables the PEN read function. The PEN function is initially off. A PEN ON statement must be executed before any pen read function calls can be made. A call to the PEN function while the PEN function is off results in an **Illegal function call** error.

Conversely, to improve execution speed, turn the pen off with a PEN OFF statement when you are not using the light pen.

PEN
Statement and Function

For Advanced BASIC, executing PEN ON also allows trapping to take place with the ON PEN statement. After PEN ON, if a nonzero line number was specified in the ON PEN statement, then every time the program starts a new statement BASIC checks to see if the pen was activated. See "ON PEN Statement."

PEN OFF disables the PEN read function. For Advanced BASIC, no trapping of the pen takes place. Action by the light pen is not remembered even if it does take place.

PEN STOP is available only in Advanced BASIC. It disables trapping of light pen activity. If activity does occur, however, it is remembered, so an immediate trap occurs when a PEN ON is executed.

When the pen is down in the border area of the screen, the values returned are inaccurate.

You should not try I/O to cassette while PEN is ON.

Example: This example prints the pen value since the last poll, and the current value.

```
10 PEN ON
20 FOR I=1 TO 500
30 X=PEN(0): X1=PEN(3)
40 PRINT X, X1
50 NEXT
600 PEN OFF
```

PLAY Statement

Purpose: Plays music as specified by *string*.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: PLAY *string*

Remarks: PLAY implements a concept similar to DRAW by imbedding a "tune definition language" into a character string.

string is a string expression consisting of single- or double-character music commands.

The commands in PLAY are:

A to G with optional #, +, or -
Plays the indicated note in the current octave. A number sign (#) or plus sign (+) afterward indicates a sharp; a minus sign (-) indicates a flat. A #, +, or - is not allowed unless it corresponds to a black key on a piano. For example, B# is an invalid note.

O n Octave. Sets the current octave for the notes that follow. There are 7 octaves, numbered Ø to 6. Each octave goes from C to B. Octave 3 starts with middle C. Octave 4 is the default octave.

> n Go up to the next higher octave and play note *n*. Each time note *n* is played, the octave goes up, until it reaches octave 6. For example, PLAY ">A" raises the octave and plays note A. Each time PLAY ">A" is executed, the octave goes up until it reaches octave 6; then

each time PLAY ">A" executes, note A plays at octave 6. (For BASIC 2.Ø and later releases.)

< n Go down one octave and play note *n*. Each time each time note *n* is played, the octave goes down, until it reaches octave Ø. For example, PLAY "<A" lowers the octave and plays note A. Each time PLAY "<A" is executed, the octave goes down until it reaches octave Ø; then each time PLAY "<A" executes, note A plays at octave Ø. (For BASIC 2.Ø and later releases.)

N n Plays note *n*, which can range from Ø to 84. In the 7 possible octaves, there are 84 notes. *n*=Ø means "rest." This is an alternative way of selecting notes besides specifying the octave (O *n*) and the note name (A-G).

L n Sets the length of the notes that follow. The actual length of the note is 1/*n*. *n* can range from 1 to 64.

Length	Equivalent
L1	whole note
L2	half note
L3	one of a triplet of three half notes (1/3 of a 4-beat measure)
L4	quarter note
L5	one of a quintuplet (1/5 of a measure)
L6	one of a quarter-note triplet
.	
.	
.	
L64	sixty-fourth note

The length can also follow the note when you want to change only the length of the note. For example, A16 is equivalent to L16A.

P n Pause (rest). *n* can range from 1 to 64, and figures the length of the pause in the same way as L (length).

. (dot or period) When placed after a note, causes the note to be played as a dotted note. A dot increases the duration of a note by half the duration of the note. A note can have more than one dot. Each dot increases the total value of the note by 1/2 the value of the previous dot. For example, a double-dotted halfnote is equivalent in duration to a half note plus a quarter note plus an eighth note. Dots can also appear after a pause (P) to scale the pause length in the same way.

T n Tempo. Sets the number of quarter notes in a minute. *n* can range from 32 to 255. The default is 12Ø. Under "SOUND Statement" is a table listing common tempos and the equivalent beats per minute.

MF Music foreground. Music (created by SOUND or PLAY) runs in foreground. Each subsequent note or sound will not start until the previous note or sound is finished. You can press Ctrl-Break to exit PLAY. Music foreground is the default state.

MB Music background. Music (created by SOUND or PLAY) runs in background instead of in foreground. Each note or sound is placed in a buffer, allowing the BASIC program to continue executing while music

plays in the background. The music background buffer can hold up to 32 notes at one time.

MN Music normal. Each note plays 7/8 of the time specified by L (length). This is the default setting of **MN, ML**, and **MS**.

ML Music legato. Each note plays the full period set by L (length).

MS Music staccato. Each note plays 3/4 of the time specified by L.

X variable;
Executes specified string.

In all these commands the *n* argument can be a constant such as **12**, or it can be ***=variable;*** where *variable* is the name of a variable. The semicolon (;) is required when you use a variable in this way, and when you use the **X** command. Otherwise a semicolon is optional between commands, except that a semicolon is not allowed after MF, MB, MN, ML, or MS. Also, any blanks in *string* are ignored.

You can also specify variables in the form VARPTR$(*variable*), instead of ***=variable;***. The VARPTR$ form is the only one that can be used in compiled programs. For example:

One Method	Alternative Method
PLAY "XA$;"	PLAY "X"+VARPTR$(A$)
PLAY "O=I;"	PLAY "O="+VARPTR$(I)

You can use **X** to store a "subtune" in one string and call it repetitively with different tempos or octaves from another string.

Examples: The following example plays a tune.

```
10 'little lamb
20 MARY$="GFE-FGGG"
30 PLAY "MB T100 O3 L8;XMARY$;P8 FFF4"
40 PLAY "GB-B-4; XMARY$; GFFGFE-...."
```

The following example plays the scale from octave Ø to octave 6.

```
10 ' Play the scale using > octave
20 SCALE$="CDEFGAB"
30 PLAY "O0 XSCALE$;"
40 FOR I=1 TO 6
50 PLAY ">XSCALE$;"
60 NEXT
70 ' Play the scale using < octave
80 PLAY "O6 XSCALE$;"
90 FOR I=1 TO 6
100 PLAY "<XSCALE$;"
110 NEXT
```

PLAY(n) Function

Purpose: Returns the number of notes currently in the music background buffer. (For BASIC 2.Ø and later releases.)

Versions: Cassette Disk Advanced Compiler
***

Format: *v*=PLAY(*n*)

Remarks:

n is a dummy argument that can have any value.

PLAY(*n*) returns a Ø when the program is running in Music Foreground mode. The maximum value that can be returned is 32, which is the maximum number of notes held in the buffer.

PLAY(n) returns notes in the buffer only when you are using Music Background (MB) mode.

Example:

```
1Ø 'When 5 notes are left in the background music buffer
2Ø 'go to line 1ØØØ and play another tune
3Ø PLAY "MB CDEFGAB"
4Ø IF PLAY(1)=5 GOTO 1ØØØ
5Ø GOTO 2ØØØ
   .
   .
   .
1ØØØ PLAY "MB Ø4 T2ØØ L4 MS GG#GE"
2ØØØ END
```

PMAP
Function

Purpose: Maps physical coordinates to world coordinates or world coordinates to physical coordinates. (For BASIC 2.Ø and later releases.)

Versions: Cassette Disk Advanced Compiler
***

Graphics mode only

Format: *v*=PMAP(*x*,*n*)

Remarks:

x coordinate of the point that is to be mapped

n can be a value in the range Ø to 3 such that:

Ø maps the world coordinate *x* to the physical coordinate *x*

1 maps the world coordinate *y* to the physical coordinate *y*

2 maps the physical coordinate *x* to the world coordinate *x*

3 maps the physical coordinate *y* to the world coordinate *y*

PMAP is used to translate coordinates between the world system as defined by the WINDOW statement and the physical coordinate system.

PMAP Function

PMAP(x,Ø) and PMAP(x,1) are used to map values from the world coordinate system to the physical coordinate system.

PMAP(x,2) and PMAP(x,3) are used to map values from the physical coordinate system to the world coordinate system.

For example, if the statement

```
SCREEN 1: WINDOW (-1,-1)-(1,1)
```

is in effect you can use PMAP to map the world coordinate points of (-1,-1) and (1,1) to their corresponding physical points on the screen.

PMAP(-1,Ø) returns the physical x coordinate value of Ø.

PMAP(-1,1) returns the physical y coordinate value of 199.

PMAP(1,Ø) returns the physical x coordinate value of 319.

PMAP(1,1) returns the physical y coordinate value of Ø.

The above information tells you that the point (-1,-1), which is in the lower left corner of the screen, corresponds to the physical point (Ø,199). You also know that the point (1,1), which is in the upper right corner, corresponds to the physical point (319,Ø).

POINT
Function

Purpose: The first form returns the attribute of the specified point on the screen. The second form returns the value of the current x or y graphics coordinate.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Graphics mode only.

Format: *v* = POINT (*x,y*)

v = POINT (*n*)

Remarks:

(x,y) are the coordinates of the point to be used. They must be in absolute form as explained in "Specifying Coordinates" under "Graphics Modes" in Chapter 3 of the *BASIC Handbook*.

If the point given is out of range, the value -1 is returned. In medium resolution valid returns are Ø, 1, 2, and 3. In high resolution they are Ø and 1.

n returns the value of the current *x* or *y* graphics coordinate. (For BASIC 2.Ø and later releases.) *n* can have a value from Ø to 3 where:

Ø returns the current physical *x* coordinate.

1 returns the current physical *y* coordinate.

2	returns the current world *x* coordinate if WINDOW is active. If WINDOW is not active, returns the current physical *x* coordinate.
3	returns the current world *y* coordinate if WINDOW is active. If WINDOW is not active, returns the current physical *y*.

See also "WINDOW Statement."

Example: The following example inverts the current setting of point (I,I).

```
10 SCREEN 2
20 IF POINT(I,I)<>0 THEN PRESET(I,I)
                    ELSE PSET(I,I)
   or
20 PSET(I,I),1-POINT(I,I)
```

POINT
Function

This example illustrates values returned by the POINT function. Note the change in the values depending upon WINDOW.

```
10 CLS:SCREEN 1,0:KEY OFF
20 PRINT "POINT(n) with WINDOW inactive"
30 GOSUB 110
40 WINDOW (0,0)-(319,199)
50 PRINT "POINT(n) with WINDOW active"
60 GOSUB 110
70 PRINT "POINT(n) with WINDOW and SCREEN active"
80 WINDOW SCREEN (0,0)-(319,199)
90 GOSUB 110
100 END
110 PSET (5,15)
120 FOR I=0 TO 3
130 PRINT POINT (I);
140 NEXT
150 PRINT:PRINT
160 RETURN
RUN

POINT(n) with WINDOW inactive
5  15  5  15
POINT(n) with WINDOW active
5  184  5  15
POINT(n) with WINDOW and SCREEN active
5  15  5  15
```

POKE Statement

Purpose: Writes a byte into a memory location.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: POKE *n,m*

Remarks:

n must be in the range Ø to 65535. It indicates the offset into the current segment where that data is to be written. The current segment is defined by the DEF SEG statement. See "DEF SEG Statement."

m m is the data to be written to the specified location. It must be in the range Ø to 255.

The complementary function to POKE is PEEK. POKE and PEEK are useful for efficient data storage and loading assembly language subroutines. See also "Peek Function."

Warning:
BASIC does not check the offset specified. So don't go POKEing around in BASIC's stack, BASIC's variable area, or your BASIC program.

Example: See the examples in Appendix B, "Assembly Language Subroutines."

POS
Function

Purpose: Returns the current cursor column position.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: v = POS(n)

Remarks: n is a dummy argument.

The current horizontal (column) position of the cursor is returned. The returned value is in the range 1 to 4Ø or 1 to 8Ø, depending on the current WIDTH setting. CSRLIN can be used to find the vertical (row) position of the cursor. See "CSRLIN Variable."

See also "LPOS Function."

Example: This example prints a carriage return (moves the cursor to the beginning of the next line) if the cursor is beyond position 6Ø on the screen.

```
IF POS(Ø)>6Ø THEN PRINT CHR$(13)
```

PRINT Statement

Purpose: Displays data on the screen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: PRINT [*list of expressions*] [;]

? [*list of expressions*] [;]

Remarks:

list of expressions
is a list of numeric and/or string expressions, separated by commas, blanks, or semicolons. Any string constants in the list must be enclosed in quotation marks.

Remarks:

If the list of expressions is omitted, a blank line is displayed. If the list of expressions is included, the values of the expressions are displayed on the screen.

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC divides the line into print zones of 14 spaces each.

In the list of expressions:

- Typing a comma between expressions causes the next value to be printed at the beginning of the next zone.

- Typing a semicolon causes the next value to be printed immediately after the last value.

- Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma, semicolon, or SPC or TAB function ends the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions ends without a comma, semicolon, SPC or TAB function, a carriage return is printed at the end of the line; that is, BASIC moves the cursor to the beginning of the next line.

If the length of the value to be printed exceeds the number of character positions remaining on the current line, the value is printed at the beginning of the next line. If the value to be printed is longer than the defined WIDTH, BASIC prints as much as it can on the current line and continues printing the rest of the value on the next physical line.

Scrolling occurs as described under "Text Mode" in Chapter 3 of the *BASIC Handbook*.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. When single-precision numbers can be represented with 7 or fewer digits in fixed-point format as accurately as in floating point-format, they are returned in fixed-point or integer format. For example, 1Ø∧(-7) is printed as .ØØØØØØ1 and 1Ø∧(-8) is output as 1E-8.

BASIC automatically inserts a carriage return/line feed after printing *width* characters, where *width* is 4Ø or 8Ø, as defined by the WIDTH statement. This

causes two lines to be skipped when you print exactly 4Ø (or 8Ø) characters, unless the PRINT statement ends in a semicolon (;).

LPRINT is used to print information on the printer. See "LPRINT and LPRINT USING Statements."

PRINT Statement

Example: In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

```
10 X=5
20 PRINT X+5, X-5, X*(-5)
RUN
 10            0            -25
```

Here, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line.

```
10 INPUT X
20 PRINT X;"SQUARED IS";X^2;"AND";
30 PRINT X;"CUBED IS";X^3
RUN
 ?9
 9 SQUARED IS 81 AND 9 CUBED IS 729
```

PRINT USING Statement

Purpose: Prints strings or numbers using a specified format.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: PRINT USING *v$; list of expressions* [;]

Remarks:

v$ is a string constant or variable that consists of special formatting characters. These formatting characters determine the field and the format of the printed strings or numbers.

list of expressions
consists of the string or numeric expressions that are to be printed, separated by semicolons or commas.

String Fields

When PRINT USING is used to print strings, one of three formatting characters can be used to format the string field:

! Specifies that only the first character in the given string is to be printed.

n spaces\ Specifies that 2 + *n* characters from the string are to be printed. If the backslashes are typed with no spaces, two characters are printed; with one space, three characters are printed, and so on. If the string is longer than the

field, the extra characters are ignored. If the field is longer than the string, the string is left-justified in the field and padded with spaces on the right.

Example: This example shows how to use *!* and \\ to print string fields.

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;B$
30 PRINT USING "\    \";A$;B$
RUN
LO
LOOKOUT
```

& Specifies a variable-length string field. When the field is specified with "&", the string is output exactly as input. Example:

```
10 A$="LOOK": B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters can be used to format the numeric field:

A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number is right-justified (preceded by spaces) in the field.

A decimal point can be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit is always printed (as Ø if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
 Ø.78
```

Example: In this example, three spaces are inserted at the end of the format string to separate the printed values on the line.

```
PRINT USING "##.##   ";1Ø.2,5.3,66.789,.234
1Ø.2Ø    5.3Ø    66.79    Ø.23
```

+ A plus sign at the beginning or end of the format string causes the sign of the number (plus or minus) to be printed before or after the number.

```
PRINT USING "+##.##   ";-68.95,2.4,55.6,-.9
-68.95    +2.4Ø   +55.6Ø    -Ø.9Ø
```

- A minus sign at the end of the format field causes negative numbers to be printed with a trailing minus sign.

```
PRINT USING "##.##-  ";-68.95,22.449,-7.Ø1
68.95-   22.45     7.Ø1-
```

** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

```
PRINT USING "**#.#   ";12.39,-Ø.9,765.1
*12.4   *-Ø.9   765.1
```

$$ A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$.

```
PRINT USING "$$###.##   ";456.78,Ø.9,-765.1
 $456.78     $Ø.9Ø      -$765.1Ø
```

***$* The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces are filled with asterisks, and a dollar sign is printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING "**$##.##";2.34
***$2.34
```

, A comma left of the decimal point in a formatting string prints a comma left of every third digit left of the decimal point. A comma at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (^^^^) format.

```
PRINT USING "####,.##";1234.5
1,234.5Ø

PRINT USING "####.##,";1234.5
1234.5Ø,
```

^^^^ Four carets can be placed after the digit position characters to specify exponential format. The four carets allow space for E±nn or D±nn to be printed. Any decimal point position can be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position is used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.##^^^^";234.56
 2.35EØ2

PRINT USING ".###^^^^-";-88888
.889EØ5-

PRINT USING "+.##^^^^";123
+.12EØ3
```

_ An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "_!##.##_!";12.34
!12.34!
```

The literal character itself can be an underscore by placing "____" in the format string.

If the number to be printed is larger than the specified numeric field, a percent sign (%) is printed in front of the number. If rounding causes the number to exceed the field, the percent sign is printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22

PRINT USING ".##";.999
%1.ØØ
```

If the number of digits specified exceeds 24, an **Illegal function call** error occurs.

Example: This example shows how you can include string constants in the format string.

```
PRINT USING "THIS IS EXAMPLE _##"; 1
THIS IS EXAMPLE #1
```

PRINT # and PRINT # USING Statements

Purpose: Writes data sequentially to a file.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: PRINT #*filenum*, [USING *x$*;] *list of exps* [;]

Remarks:

filenum is the number used when the file was opened for output.

x$ is a string expression comprised of formatting characters as described in the previous entry, "PRINT USING Statement."

list of exps is a list of the numeric and/or string expressions that will be written to the file.

PRINT # does not compress data in the file. An image of the data is written to the file just as it would be displayed on the screen with a PRINT statement. For this reason, care should be taken to delimit the data in the file, so that it is input correctly from the file.

In the list of expressions, numeric expressions should be delimited by semicolons. For example,

```
PRINT #1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks inserted between print fields are also written to the file.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly in the file, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="936Ø4-1". The statement

```
PRINT #1,A$;B$
```

writes CAMERA936Ø4-1 to the file. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT # statement as follows:

```
PRINT #1,A$;",";B$
```

The image written to the file is

```
CAMERA,936Ø4-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to the file surrounded by explicit quotation marks using CHR$(34).

For example, let A$="CAMERA, AUTOMATIC" and B$=" 936Ø4-1". The statement:

```
PRINT #1,A$;B$
```

writes the following image to the file:

```
CAMERA, AUTOMATIC   936Ø4-1
```

and the statement:

```
INPUT #1,A$,B$
```

inputs the string "CAMERA" to A$ and "AUTOMATIC 936Ø4-1" to B$.

To separate these strings properly in the file, write double quotes to the file image using CHR$(34). The statement:

```
PRINT #1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to the file:

```
"CAMERA, AUTOMATIC" "936Ø4-1"
```

and the statement:

```
INPUT #1,A$,B$
```

inputs "CAMERA, AUTOMATIC" to A$ and "936Ø4-1" to B$.

The PRINT # statement can also be used with the USING option to control the format of the file. For example:

```
PRINT #1,USING"$$###.##,";J;K;L
```

PRINT # and PRINT # USING Statements

Example: Since data written to the file contains a dollar sign, use string variables to read them back, as in this example.

```
10 A=123
20 B=6789
30 C=22.33
40 OPEN "DATA" FOR OUTPUT AS #1
50 PRINT #1,USING "$$###.##,";A;B;C
60 CLOSE
70 OPEN "DATA" FOR INPUT AS #1
80 INPUT #1,A$,B$,C$
90 CLOSE
100 PRINT A$,B$,C$
```

PSET and PRESET Statements

Purpose: Draws a point at the specified position on the screen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Graphics mode only.

Format: PSET (*x,y*) [,*color*]

PRESET (*x,y*) [,*color*]

Remarks:

(x,y) are the coordinates of the point to be set. They can be in absolute or relative form, as explained in "Specifying Coordinates" under "Graphics Modes" in Chapter 3 of the *BASIC Handbook*.

color is an integer expression that chooses an attribute from the attribute range for the current screen mode. In medium resolution, the color is the current one for that attribute as defined by the COLOR statement. Four attributes (Ø-3) are available in medium resolution; in high resolution, two attributes (Ø-1) are available. Zero (Ø) is always the attribute for the background. The default foreground attribute is always the maximum attribute for that screen mode: 3 in medium resolution; 1 in high resolution.

PRESET is almost identical to PSET. The only difference is that if no *color* parameter is given to

PRESET, the background attribute (Ø) is selected. If *color* is included, PRESET is identical to PSET. Line 7Ø in the example below could just as easily be:

```
7Ø PSET(I,I),Ø
```

Out-of-range coordinates are clipped.

Example: Lines 2Ø–4Ø of this example draw a diagonal line from the point (Ø,Ø) to the point (1ØØ,1ØØ). Then lines 6Ø–8Ø erase the line by setting each point to a color of Ø.

```
1Ø CLS:SCREEN 1:KEY OFF
2Ø FOR I=Ø TO 1ØØ
3Ø PSET (I,I)
4Ø NEXT
5Ø 'erase line
6Ø FOR I=1ØØ TO Ø STEP -1
7Ø PRESET(I,I)
8Ø NEXT
```

PUT
Statement (Files)

Purpose: Writes a record from a random buffer to a random file.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: PUT [#]*filenum* [,*number*]

Remarks:

filenum is the number under which the file was opened.

number is the record number for the record to be written, in the range 1 to 16 megabytes.

If *number* is omitted, the record has the next available record number (after the last PUT).

PRINT #, PRINT # USING, WRITE #, LSET, and RSET can be used to put characters in the random file buffer before a PUT statement. In the case of WRITE #, BASIC pads the buffer with spaces up to the carriage return.

Any attempt to read or write past the end of the buffer causes a **Field overflow** error. See also Appendix A, "BASIC Disk Input and Output," in the *BASIC Handbook*.

Because BASIC and DOS block as many records as possible in 512-byte sectors, the PUT statement does not necessarily perform a physical write to the disk for each record.

PUT
Statement (Files)

PUT can be used for a communications file. In that case *number* is the number of bytes to write to the communications file. This number must be less than or equal to the value set by the LEN option on the OPEN "COM... statement.

Example: See Appendix A, "BASIC Disk Input and Output," in the *BASIC Handbook*.

PUT
Statement (Graphics)

Purpose: Plots images on a specified area of the screen.

Versions:

Cassette	Disk	Advanced	Compiler
		***	***

Graphics mode only.

Format: PUT (*x,y*), *array* [,*action*]

Remarks:

(x,y) are the coordinates of the top left corner of the image to be transferred.

array is the name of a numeric array containing the information to be transferred. For more information on this array, see also "GET Statement (Graphics)."

action is one of:

PSET
PRESET
XOR
OR
AND

XOR is the default.

PUT is the opposite of GET in the sense that it takes data out of the array and puts it on the screen. However, it also provides the option of interacting with the data already on the screen.

PSET simply stores the data from the array onto the screen, so this is the true opposite of GET.

PRESET is the same as PSET except that a complementary image is produced. For example, in medium resolution, which has a maximum attribute of 3, an attribute of Ø in the array causes the corresponding point to be plotted with an attribute of 3, and vice versa; an attribute of 1 in the array causes the corresponding point to be plotted with an attribute of 2, and vice versa.

AND, OR, and XOR specify the logical operations on the bits of each image. AND is used when an image already exists in the area to which the image is transferred.

OR is used to superimpose the transferred image onto the existing image.

XOR is a special mode that can be used for animation. Its unique property is that when an image is PUT against a complex background *twice*, the background is restored unchanged. This allows you to move an object around without obliterating the background.

In medium resolution mode, AND, XOR, and OR have the following effects on color:

AND

screen	array value Ø	1	2	3
Ø	Ø	Ø	Ø	Ø
1	Ø	1	Ø	1
2	Ø	Ø	2	2
3	Ø	1	2	3

OR

screen	array value Ø	1	2	3
Ø	Ø	1	2	3
1	1	1	3	3
2	2	3	2	3
3	3	3	3	3

XOR

screen	array value Ø	1	2	3
Ø	Ø	1	2	3
1	1	Ø	3	2
2	2	3	Ø	1
3	3	2	1	Ø

Animation of an object can be performed as follows:

1. PUT the object on the screen (with XOR).
2. Recalculate the new position of the object.
3. PUT the object on the screen (with XOR) a second time at the old location to remove the old image.
4. Repeat step 1, this time putting the object at the new location.

Movement done this way leaves the background unchanged. Flicker can be reduced by minimizing the time between steps 4 and 1, and making sure there is enough time delay between steps 1 and 3. If more than one object is being animated, each object should be processed individually, one step at a time.

If it is not important to preserve the background, animation can be performed using the PSET action verb. But you should remember to have an image area that will contain the "before" and "after"

images of the object. This way the extra area effectively erases the old image. This method can be somewhat faster than the method using XOR described above, since only one PUT is required to move an object (although you must PUT a larger image).

If the image to be transferred is too large to fit on the screen, an **Illegal function call** error occurs.

Example: This example shows how to move a circle across the screen with XOR.

```
10 CLS:DEFINT A-Z:SCREEN 1:KEY OFF
20 DIM A(404)
30 CIRCLE (160,100), 20,3
40 PAINT (160,100),2,3
50 GET (140,80)-(180,120),A:CLS
60 X=30: Y=50
70 FOR I=1 TO 20
80 PUT (X,Y),A,XOR
90 PUT (X,Y),A,XOR
100 X=X + 10
110 NEXT
```

RANDOMIZE
Statement

Purpose: Reseeds the random number generator.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: RANDOMIZE [*n*]

RANDOMIZE TIMER

Remarks:

n is an integer, single-, or double-precision expression that is used as the random number seed. In Cassette BASIC, *n* must be an integer expression.

If *n* is omitted, BASIC suspends program execution and asks for a value by displaying:

```
Random Number Seed (-32768 to 32767)?
```

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers every time the program is run, place a RANDOMIZE statement at the beginning of the program and change the seed with each run.

In Disk BASIC and Advanced BASIC, the internal clock can be a useful way to get a random number seed. You can use VAL to change the last two digits of TIME$ to a number, and then use that number for *n*.

You can get a new random number seed without being prompted. To do this, use the TIMER function in the expression. (For BASIC 2.Ø and later releases.)

Example:

```
1Ø RANDOMIZE
2Ø FOR I=1 TO 4
3Ø PRINT RND;
4Ø NEXT I

RUN
Random Number Seed (-32768 to 32767)?
```

Suppose you respond with 3. The program continues:

```
Random Number Seed (-32768 to 32767)? 3
 .7655695 .3558Ø7 .3742327 .1388798
RUN
Random Number Seed (-32768 to 32767)?
```

Suppose this time you respond with 4. The program continues:

```
Random Number Seed (-32768 to 32767)? 4
 .1719568 .5273236 .6879686 .713297
RUN
Random Number Seed (-32768 to 32767)?
```

If you try 3 again, you'll get the same sequence as the first run:

```
Random Number Seed (-32768 to 32767)? 3
 .7655695 .3558Ø7 .3742327 .1388798
```

RANDOMIZE Statement

This example uses TIMER. Note that each time the program is run you see a different sequence of numbers.

```
10 RANDOMIZE TIMER
20 FOR I=1 TO 4
30 PRINT RND;
40 NEXT
RUN
 .9590051 .1036786 .1464037 .7754918
RUN
 .8261163 .17422 .9191545 .5041142
```

READ Statement

Purpose: Reads values from a DATA statement and assigns them to variables. See "DATA Statement."

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: READ *variable* [,*variable*]...

Remarks:

variable is a numeric or string variable or array element that is to receive the value read from the DATA table.

A READ statement must always be used with a DATA statement. READ statements assign DATA statement values to the variables in the READ statement on a one-to-one basis.

READ statement variables can be numeric or string, and the values that are read must agree with the variable types specified. If they do not agree, a **Syntax error** results.

A single READ statement can access one or more DATA statements (they are accessed in order), or several READ statements can access the same DATA statement. If the number of variables in the list of variables exceeds the number of elements in the DATA statement(s), an **Out of data** error occurs. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

READ Statement

To reread data from any line in the list of DATA statements, use the RESTORE statement. See "RESTORE Statement."

Example: This program segment reads the values from the DATA statements into the array A. After execution, the value of A(1) is 3.Ø8, and so on.

```
1Ø FOR I=1 TO 1Ø
2Ø READ A(I)
3Ø NEXT I
4Ø DATA 3.Ø8,5.19,3.12,3.98,4.24
5Ø DATA 5.Ø8,5.55,4.ØØ,3.16,3.37
```

This program reads string and numeric data from the DATA statement in line 3Ø. Note that you do not need quotation marks around COLORADO, because it does not have commas, semicolons, or significant leading or trailing blanks. However, you do need the quotation marks around "DENVER," because of the comma.

```
1Ø PRINT "CITY", "STATE", " ZIP"
2Ø READ C$,S$,Z
3Ø DATA "DENVER,", COLORADO, 8Ø211
4Ø PRINT C$,S$,Z
RUN
CITY          STATE          ZIP
DENVER,       COLORADO       8Ø211
```

REM Statement

Purpose: Inserts explanatory remarks in a program.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: REM *remark*

Remarks: *remark* can be any sequence of characters.

REM statements are not executed, but are displayed when the program is listed exactly as they were entered. However, they do slow execution time somewhat and take up space in memory.

REM statements can be branched into (from a GOTO or GOSUB statement), and execution continues with the first executable statement after the REM statement.

Remarks can be added by preceding the remark with a single quotation mark instead of :REM. If you put a remark on a line with other BASIC statements, the remark must be the *last* statement on the line.

You cannot use the single quote (') to add comments at the end of a DATA statement. If you do, BASIC thinks it is part of a string. You can, however, use :REM to add a remark.

REM Statement

Example: This example shows the two ways to insert remarks in a program.

```
10 'calculate average velocity
20 SUM=0: REM initialize SUM
30 FOR I=1 TO 20
40 SUM=SUM + V(I)
```

Line 20 might also be written:

```
20 SUM=0 ' initialize SUM
```

RENUM Command

Purpose: Renumbers program lines.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: RENUM [*newnum*] [,[*oldnum*] [,*increment*]]

Remarks:

newnum is the first line number to be used in the new sequence. The default is 1Ø.

oldnum is the line in the current program where renumbering is to begin. The default is the first line of the program.

increment is the amount each line number will increase in the new sequence. The default is 1Ø.

To reflect the new line numbers, RENUM also changes all line number references following ELSE, GOSUB, GOTO, ON...GOTO, ON...GOSUB, RESTORE, RESUME, THEN and ERL test statements. If a nonexistent line number appears after one of these statements, the error message **Undefined line number xxxxx in yyyyy** is displayed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyyy may be changed.

Note: RENUM cannot be used to change the order of program lines or to create line numbers greater than 65529. An attempt to do so results in an **Illegal function call** error.

RENUM Command

Example: This example renumbers the entire program. The first new line number is 1Ø. Line numbers increment by 1Ø.

```
RENUM
```

This example also renumbers the entire program. The first new line number is 3ØØ. Line numbers increment by 5Ø.

```
RENUM 3ØØ,,5Ø
```

This example renumbers the lines from 9ØØ up so they start with line number 1ØØØ and increment by 2Ø.

```
RENUM 1ØØØ,9ØØ,2Ø
```

RESET Command

Purpose: Closes all disk files and clears the system buffer.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: RESET

Remarks: If all open files are on disk, RESET is the same as CLOSE with no file numbers after it.

RESTORE
Statement

Purpose: Allows DATA statements to be reread from a specified line.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: RESTORE [*line*]

Remarks:

line is the line number of a DATA statement in the program.

After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If *line* is specified, the next READ statement accesses the first item in the specified DATA statement.

Example: In this example, the RESTORE statement in line 2Ø resets the DATA pointer to the beginning so that the values that are read in line 3Ø are 57, 68, and 79.

```
1Ø READ A,B,C
2Ø RESTORE
3Ø READ D,E,F
4Ø DATA 57, 68, 79
5Ø PRINT A;B;C;D;E;F
RUN
 57  68  79  57  68  79
```

RESUME Statement

Purpose: Continues program execution after an error recovery procedure is performed.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: RESUME [Ø]

RESUME NEXT

RESUME *line*

Remarks: Any of the formats shown above can be used, depending on where execution is to resume:

RESUME or RESUME Ø
Execution resumes at the statement that caused the error.

> **Note:** If you try to renumber a program containing a RESUME Ø statement, you will get an **Undefined line number** error. The statement will still say RESUME Ø, which is okay.

RESUME NEXT
Execution resumes at the statement immediately following the one that caused the error.

RESUME line Execution resumes at the specified line number.

RESUME Statement

A RESUME statement that is not in an error trap routine causes a **RESUME without error** message to occur.

Example: In this example, line 1ØØØ is the beginning of the error trapping routine. The RESUME statement causes the program to return to line 8Ø when error 23Ø occurs in line 9Ø.

```
1Ø ON ERROR GOTO 1ØØØ
   .
   .
   .
1ØØØ IF (ERR=23Ø)AND(ERL=9Ø) THEN PRINT
            "TRY AGAIN": RESUME 8Ø
```

Purpose: Stops a subroutine and returns to the main program. See "GOSUB and RETURN Statements."

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: RETURN [*line*]

Remarks:

line is the number of the program line you wish to return to. You can use it only in Advanced BASIC and BASIC Compiler.

Although you can use RETURN *line* to return from any subroutine, this enhancement was added to allow nonlocal returns from the event trapping routines. From one of these routines you will often want to go back to the BASIC program at a fixed line number while still eliminating the GOSUB entry the trap created. The nonlocal RETURN must be used with care, however, since any other GOSUB, WHILE, or FOR statements active at the time of the trap remain active.

RIGHT$
Function

Purpose: Returns the rightmost *n* characters of string *x$*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = RIGHT$(*x$*,*n*)

Remarks:

x$ is any string expression.

n is an integer expression that specifies the number of characters to be in the result.

If *n* is greater than or equal to LEN(*x$*), then *x$* is returned. If *n* is zero, the null string (length zero) is returned.

See also "MID$" and "LEFT$" functions.

Example: In this example, the rightmost seven characters of the string A$ are returned.

```
1Ø A$="BOCA RATON, FLORIDA"
2Ø PRINT RIGHT$(A$,7)
RUN
FLORIDA
```

Purpose: Removes a directory from the specified disk. (For BASIC 2.Ø and later releases.)

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	

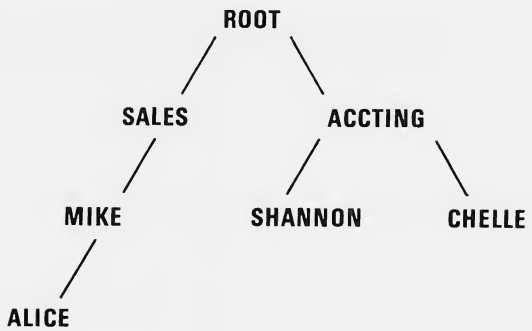
Format: RMDIR *path*

Remarks:

path is a string expression, not exceeding 63 characters, that identifies the subdirectory to be removed from the existing directory. See also "Naming Files" and "Tree-Structured Directories" in Chapter 3 of the *BASIC Handbook* for more information.

The directory must be empty of all files and subdirectories before it can be removed, with the exception of the "." and ".." entries, or a **Path/file access** error occurs.

Example:



RMDIR Command

The examples that follow refer to the tree structure shown on the preceding page.

If you are in the root directory and you want to remove the directory called ALICE, use:

```
RMDIR  "SALES\MIKE\ALICE"
```

If you want to make ACCTING the current directory and remove the directory called CHELLE, use:

```
CHDIR "ACCTING"
RMDIR "CHELLE"
```

Another way to remove the directory CHELLE is to make the root the current directory and then remove CHELLE.

```
CHDIR "\"
RMDIR "ACCTING\CHELLE"
```

The directory preceding the current directory cannot be removed. Using the same tree structure, suppose that MIKE is the current directory. If you try to remove the SALES directory, you will get a **Path/file access** error.

If you try to use the KILL command to remove a directory, you will get a **Path/file access** error.

RND Function

Purpose: Returns a random number between Ø and 1.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: v = RND[(x)]

Remarks:

x is a numeric expression that affects the returned value as described below.

The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded. Reseeding is most easily done by using the RANDOMIZE statement. See "RANDOMIZE Statement."

You can also reseed the generator when you call the RND function by using x where x is negative. This always generates the particular sequence for the given x. This sequence is not affected by RANDOMIZE, so if you want it to generate a different sequence each time the program is run, you must use a different value for x each time.

If x is positive or not included, RND(x) generates the next random number in the sequence.

RND(Ø) repeats the last number generated.

To get random numbers in the range Ø (zero) through n, use the formula:

```
INT(RND * (n+1))
```

RND
Function

Example: In this example, the first horizontal line of results shows three random numbers, generated using a positive *x*.

In line 4Ø, a negative number is used to reseed the random number generator. The random numbers produced after this reseeding are in the second row of results.

In line 8Ø, the random number generator is reseeded using the RANDOMIZE statement; in line 9Ø it is reseeded again by calling RND with the same negative value as in line 4Ø. This cancels the effect of the RANDOMIZE statement, as you can see; the third line of results is identical to the second line.

In line 13Ø, RND is called with an argument of Ø, so the last number printed is the same as the preceding number.

```
1Ø FOR I=1 TO 3
2Ø PRINT RND(I);     ' x>Ø
3Ø NEXT I
4Ø PRINT: X=RND(-6) ' x<Ø
5Ø FOR I=1 TO 3
6Ø PRINT RND(I);     ' x>Ø
7Ø NEXT I
8Ø RANDOMIZE 853     ' randomize
9Ø PRINT: X=RND(-6) ' x<Ø
1ØØ FOR I=1 TO 3
11Ø PRINT RND;       ' same as x>Ø
12Ø NEXT I
13Ø PRINT: PRINT RND(Ø)
RUN
 .6291626  .1948297  .63Ø5799
 .6818615  .4193624  .6215937
 .6818615  .4193624  .6215937
 .6215937
```

Reseeding with the RND (negative number) function reseeds through permutation of the last floating point temporary. Since no floating point calculations are done in this example, the new seed is always the same.

```
1Ø DEFINT A-Z
2Ø FOR J=1 TO 5
3Ø X=RND(-J)
4Ø FOR I=1 TO 5:PRINT RND;NEXT:PRINT
5Ø NEXT J
```

If line 2Ø is changed to:

```
2Ø FOR J=1 TO 2 STEP .1
```

a new seed is generated each time.

RUN
Command

Purpose: Begins execution of a program.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: RUN [*line*]

RUN *filespec*[,R]

Remarks:

line is the line number of the program in memory where you want execution to begin.

filespec is a string expression for the file specification. In BASIC 2.0 and later releases, it can contain a path. *Filespec* must conform to the rules outlined under "Naming Files" in Chapter 3 of the *BASIC Handbook*; otherwise, an error occurs.

RUN or RUN *line* begins execution of the program currently in memory. If *line* is specified, execution begins with the specified line number. Otherwise, execution begins at the lowest line number.

RUN *filespec* loads a file from disk or cassette into memory and runs it. It closes all open files and deletes the current contents of memory before loading the designated program. However, with the **R** option, all data files remain open. See also Appendix A, "BASIC Disk Input and Output" in the *BASIC Handbook*.

Executing a RUN command turns off any sound that is running and resets to Music Foreground. Also, PEN and STRIG are reset to OFF.

Example: The first example shows the first form of RUN in two very short programs. The first program is run from the beginning. The RUN *line* option in the second example runs the program from line 2Ø. In this case, line 1Ø is not executed, so PI does not receive its proper value. A Ø is printed because all numeric variables have an initial value of zero.

```
1Ø PRINT 1/7
RUN
 .1428571

1Ø PI=3.141593
2Ø PRINT PI
RUN 2Ø
 Ø
```

This example loads the program "NEWFIL" and runs it, keeping files open.

```
RUN "NEWFIL",R
```

SAVE Command

Purpose: Saves a BASIC program file on disk or cassette.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: SAVE *filespec* [,A]

SAVE *filespec* [,P]

Remarks:

filespec is a string expression for the file specification. In BASIC 2.0 and later releases, it can contain a path. *Filespec* must conform to the rules outlined under "Naming Files" in Chapter 3 of the *BASIC Handbook*; otherwise, an error occurs.

The BASIC program is written to the specified device. When the program is being saved to CAS1:, the cassette motor is turned on and the file is immediately written to the tape.

For disk files, if the filename is eight characters or less and no extension is supplied, the extension .BAS is added to the name. If a file with the same filename already exists on the diskette, it will be written over.

In Cassette BASIC, if the device name is omitted, CAS1: is assumed. CAS1: is the only device allowed for SAVE in Cassette BASIC.

For Disk BASIC and Advanced BASIC, the device defaults to the DOS default drive.

The **A** option saves the program in ASCII format. Otherwise, BASIC saves the file in a compressed binary (tokenized) format. ASCII files take up more space, but some types of access require that files be in ASCII format. For example, a file intended to be merged must be saved in ASCII format. Programs saved in ASCII can be read as data files.

The **P** (protection) option saves the program in an encoded binary format. When a protected program is later run (or loaded), any attempt to LIST or EDIT it fails with an **Illegal function call** error. No way is provided to "unprotect" such a program.

> **Note:** The disk directory entry for a BASIC program file gives no indication that the file is either protected or stored in ASCII format. The .BAS extension is used in any case.

See also Appendix A, "BASIC Disk Input and Output," in the *BASIC Handbook*.

Example: This example saves the program in memory as INVENT with the default extension .BAS.

```
SAVE "INVENT"
```

This example saves PROG.BAS on drive B in ASCII, so it can be merged later.

```
SAVE "B:PROG",A
```

This example saves SECRET.BOZ on drive A, protected, so it cannot be altered.

```
SAVE "A:SECRET.BOZ",P
```

SCREEN
Function

Purpose: Returns the ASCII code (Ø-255) for the character on the active screen at the specified row (line) and column.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = SCREEN(*row,col*[,*z*])

Remarks:

row is a numeric expression in the range 1 to 25.

col is a numeric expression in the range 1 to 4Ø or 1 to 8Ø, depending on the WIDTH setting.

z is a numeric expression that evaluates to a true or false value. *z* is valid only in text mode.

For a list of ASCII codes, See Appendix D, "ASCII Character Codes."

In text mode, if *z* is included and is true (nonzero), the color attribute for the character is returned instead of the code for the character. The color attribute is a number in the range Ø to 255. This number, *v*, is deciphered as follows:

(*v* MOD 16) is the foreground attribute.

(((*v* - *foreground*)/16) MOD 128) is the background attribute, where *foreground* is calculated as above.

(v>127) is true (-1) if the character is blinking; false (Ø) if it is not.

For a list of colors and their associated attributes, see "COLOR Statement."

In graphics mode, if the specified location contains graphic information (points or lines, not just a character), then the SCREEN function returns Ø.

Any values entered outside the ranges indicated result in an **Illegal function call** error.

The SCREEN *statement* is explained in the next entry.

Example: In this example, if the character at 1Ø,1Ø is A, then X is 65.

```
1ØØ X = SCREEN (1Ø,1Ø)
```

This example returns the color attribute of the character in the upper left-hand corner of the screen.

```
11Ø X = SCREEN (1,1,1)
```

SCREEN Statement

Purpose: Sets the screen attributes to be used by subsequent statements.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Meaningful with the Color/Graphics Monitor Adapter only.

Format: SCREEN [*mode*] [,[*burst*] [,[*apage*] [,*vpage*]]]

Remarks:

mode is a numeric expression resulting in an integer value of Ø, 1, or 2. Valid modes are:

- **Ø** Text mode at current width (4Ø or 8Ø).
- **1** Medium resolution graphics mode (32Øx2ØØ). Use only with Color/Graphics Monitor Adapter.
- **2** High resolution graphics mode (64Øx2ØØ). Use only with Color/Graphics Monitor Adapter.

burst is a numeric expression resulting in a true or false value. It enables or disables color. On an RGB monitor, color burst is always on. On a composite monitor, color burst can be on or off. In text mode (*mode*=Ø), a false (zero) value disables color (only the monochrome images are displayed); a true (nonzero) value enables color (color

images are displayed). In medium resolution graphics mode (*mode*=1), a true (nonzero) value disables color, and a false (zero) value enables color. Since high resolution graphics are only two colors (black and white), this parameter has no effect in high resolution.

apage (active page) is an integer expression in the range Ø to 7 for width 4Ø;Ø to 3 for width 8Ø. It selects the page to be written to by output statements to the screen, and is valid only in text mode (*mode*=Ø).

vpage (visual page) selects the page to be displayed on the screen, in the same way as *apage* above. The visual page can be different from the active page. *vpage* is valid only in text mode (*mode*=Ø). If omitted, *vpage* defaults to *apage*.

If all parameters are valid, the new screen mode is stored; the screen is erased; the foreground color is set to white; and the background and border colors are set to black.

If the new screen mode is the same as the previous mode, nothing is changed.

If the mode is text and only *apage* and *vpage* are specified, display pages are changed for viewing. Initially, both active and visual pages default to Ø (zero). By manipulating active and visual pages, you can display one page while building another. Then you can switch visual pages instantaneously.

If you mix text and graphics in the 4Ø= or 8Ø= column graphics mode and are not using a U.S. keyboard, refer to "GRAFTABL Command" in the

Disk Operating System Reference for information regarding additional character support with the Color/Graphics monitor.

Note: Only one cursor is shared among all the pages. If you are going to switch active pages back and forth, you should save the cursor position on the current active page (using POS(Ø) and CSRLIN), before changing to another active page. Then when you return to the original page, you can restore the cursor position using the LOCATE statement.

Any parameter can be omitted. Omitted parameters, except *vpage*, assume the old value.

Any values entered outside the ranges indicated result in an **Illegal function call** error. Previous values are retained.

Example: This example selects text mode with color burst enabled, and sets active and visual page to Ø.

```
1Ø SCREEN Ø,1,Ø,Ø
```

In this example, mode and color burst remain unchanged. Active page is set to 1 and display page to 2.

```
1Ø SCREEN ,,1,2
```

This example switches to high-resolution graphics mode.

```
10 SCREEN 2,,0,0
```

This example switches to medium-resolution color graphics, color burst enabled.

```
10 SCREEN 1,0
```

This example sets medium-resolution graphics with color burst disabled.

```
50 SCREEN ,1
```

SGN
Function

Purpose: Returns the sign of x.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: v = SGN(x)

Remarks: x is any numeric expression.

SGN(x) is the mathematical signum function:

- If x is positive, SGN(x) returns 1.
- If x is zero, SGN(x) returns Ø.
- If x is negative, SGN(x) returns -1.

Example: This example branches to 1ØØ if X is negative; 2ØØ if X is zero; and 3ØØ if X is positive.

```
ON SGN(X)+2 GOTO 1ØØ,2ØØ,3ØØ
```

Purpose: Loads and executes another program file (such as files with the .COM, .BAT, and .EXE extensions). Any program executed under BASIC is referred to as a child process. When the child process has finished executing, control returns to the parent BASIC program at the statement following the SHELL statement.

(Not valid for BASIC releases earlier than 3.0.)

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	

Format: SHELL [*command string*]

Remarks:

command string is a string expression containing the name of a program to run, and, optionally, any parameters you are passing to the child process.

Child processes are executed by SHELL loading and running a copy of COMMAND.COM with the /C switch. By using COMMAND in this way any parameters you can have are correctly passed into the default File Control Blocks. Standard input and output can be redirected, and built-in commands such as DIR, PATH, and SORT can be executed. If you enter SHELL with no *command string*, a copy of COMMAND.COM is loaded, the DOS prompt appears, and you can enter any commands that are valid under DOS (DIR, COPY, SORT, ETC.). You can return to BASIC by typing the word EXIT. You can also invoke batch files from the SHELL

statement. To return to the parent BASIC program, your batch file must contain EXIT as the last statement of the batch file.

When running child processes from a BASIC application using the SHELL statement, there are some procedures and rules that your application should follow. Going outside the boundaries of these guidelines could cause your application to fail or produce unpredictable results.

To guarantee that you return to BASIC from your child process in the the screen mode that you expect, you can do one of two things:

- Use BIOS Interrupt 1ØH, function call 15, to save the current video mode. When your child process returns to DOS, use function call Ø to restore the video mode.

- From your BASIC program, execute a SCREEN statement followed by a CLS statement immediately after the SHELL statement.

Before BASIC executes a SHELL statement, it saves any interrupt vectors it uses; but this does not ensure that any interrupt vectors your routines use, and are not used by BASIC, are restored. So be sure to save any interrupt vectors your routine uses to preserve the proper interface to DOS.

Certain devices must be left untouched to ensure that they are exactly as DOS and BASIC expect them to be. These devices are the 8259 interrupt controller, the 8253 counter timer, the 8237 DMA controller, the 8255 I/O latch, and the 825Ø asynchronous communications element. Further descriptions of these devices can be found in the IBM Personal Computer *Technical Reference* manual.

A child process that alters any file opened by the BASIC application can have unpredictable results. If you must update such files, be sure to close them from your BASIC application before executing a SHELL to your child process. Remember that files that were opened under redirection of standard input and output constitute OPEN files and that these files should not be modified in a SHELLed process.

Before BASIC executes a SHELL statement, it frees any memory it is not using except when BASIC is invoked with the **/M:** switch. Because BASIC was invoked with the **/M:** switch, BASIC assumes that a assembly language routine will be loaded just beyond BASIC's data segment. This prevents BASIC from compressing its workspace before executing a SHELL and, consequently, SHELL can fail with an **Out of memory** error when using the **/M:** switch.

The preferred method for running applications that use the SHELL statement is to load the assembly language subroutines before you run BASIC. This involves placing code in your subroutines that, when invoked from DOS, allows them to terminate and stay resident (INT 27H). For more information refer to Appendix B, "Assembly Language Subroutines."

Any routine that you execute from the SHELL statement should never terminate and stay resident. Doing so may not leave BASIC enough room to restore its workspace. All files are closed, the error message **Can't continue after SHELL** is printed, and BASIC exits to DOS.

BASIC remains in memory while the child process is running. When the child process finishes, BASIC continues.

SHELL Statement

A program name in *command string* can have any extension you choose. If you do not supply an extension, DOS looks for a .COM extension, then a .EXE extension, and finally a .BAT extension. If the *filename* is not located during this search, COMMAND issues an error message.

Any text separated from a program name supplied in *command string* by at least one blank is processed as program parameters.

When BASIC is run, its environment is inherited from DOS. Any changes your application makes to BASIC's environment are reflected in the environment for the child process.

> **Note:** For more information on environment, see "Detailed Descriptions of Advanced DOS Commands" in *Disk Operating System* (DOS manual).

You *cannot* SHELL to BASIC. If you attempt to run BASIC as a child process, you will receive an error message. After this message is displayed, control returns to the BASIC parent.

Example: This example creates a file, exits to the DOS SORT utility, and then returns to BASIC.

```
10 OPEN "SORTIN.DAT" FOR OUTPUT AS #1
20 'writes data to be sorted
   .
   .
   .
100 CLOSE 1
110 SHELL "SORT <SORTIN.DAT >SORTOUT.DAT"
120 OPEN "SORTOUT.DAT" FOR INPUT AS #1
130 'processes the sorted data
```

The following example displays a disk directory from BASIC.

```
SHELL
A>DIR   (type DIR command at DOS prompt)
A>EXIT  (type EXIT to return to BASIC)
```

The same result can be achieved with:

```
SHELL "DIR".
```

SIN Function

Purpose: Calculates the trigonometric sine function.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: $v = \mathrm{SIN}(x)$

Remarks: x is an angle in radians.

To convert degrees to radians, multiply by PI/18Ø, where PI=3.141593.

In BASIC 2.Ø and later releases, you can have this calculation performed in double precision by specifying **/D** on the BASIC command line when BASIC is initially loaded. See "Options in the BASIC Command" in the *BASIC Handbook*.

Example: This example calculates the sine of 9Ø degrees after first converting the degrees to radians.

```
1Ø PI=3.141593
2Ø DEGREES = 9Ø
3Ø RADIANS=DEGREES * PI/18Ø ' PI/2
4Ø PRINT SIN(RADIANS)
RUN
 1
```

SOUND Statement

Purpose: Generates sound through the speaker.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: SOUND *freq, duration*

Remarks:

freq is the desired frequency in Hertz (cycles per second). It must be a numeric expression in the range 37 to 32767.

duration is the desired duration in clock ticks. The clock ticks occur 18.2 times per second. *duration* must be a numeric expression. In Disk BASIC, duration is .Ø27 to 65535. In Advanced BASIC, duration is .ØØ15 to 65535.

When the SOUND statement produces a sound, the program continues to execute until another SOUND statement is reached. If *duration* of the new SOUND statement is Ø, the currently running SOUND statement is turned off. Otherwise, the program waits until the first sound completes before it executes the new SOUND statement.

If you are using Advanced BASIC, you can cause the sounds to be buffered so execution does not stop when a new SOUND statement is encountered. See the explanation of the **MB** (Music Background) command under "PLAY Statement."

If no SOUND statement is running, SOUND *x*,Ø has no effect.

SOUND
Statement

The tuning note, A, has a frequency of 44Ø. The following table correlates notes with their frequencies.

Note	Frequency	Note	Frequency
C	13Ø.81Ø	C	523.25Ø
D	146.83Ø	D	587.33Ø
E	164.81Ø	E	659.26Ø
F	174.61Ø	F	698.46Ø
G	196.ØØØ	G	783.99Ø
A	22Ø.ØØØ	A	88Ø.ØØØ
B	246.94Ø	B	987.77Ø
C*	261.63Ø	C	1Ø46.5ØØ
D	293.66Ø	D	1174.7ØØ
E	329.63Ø	E	1318.5ØØ
F	349.23Ø	F	1396.9ØØ
G	392.ØØØ	G	1568.ØØØ
A	44Ø.ØØØ	A	176Ø.ØØØ
B	493.88Ø	B	1975.5ØØ

* middle C. Higher (or lower) notes can be approximated by doubling (or halving) the frequency of the corresponding note in the previous (next) octave.

To create periods of silence, use SOUND 32767,*duration*.

The duration for one beat can be calculated from beats per minute by dividing the beats per minute into 1Ø92 (the number of clock ticks in a minute).

The next table shows typical tempos in terms of clock ticks:

	Tempo	Beats/ Minute	Ticks/ Beat
very slow	Larghissimo		
	Largo	4Ø-6Ø	27.3-18.2
	Larghetto	6Ø-66	18.2-16.55
	Grave		
	Lento		
	Adagio	66-76	16.55-14.37
slow	Adagietto		
	Andante	76-1Ø8	14.37-1Ø.11
medium	Andantino		
	Moderato	1Ø8-12Ø	1Ø.11-9.1
fast	Allegretto		
	Allegro	12Ø-168	9.1-6.5
	Vivace		
	Veloce		
	Presto	168-2Ø8	6.5-5.25
very fast	Prestissimo		

Example: The following program creates a glissando (sliding up and down the scale).

```
1Ø FOR I=44Ø TO 1ØØØ STEP 5
2Ø SOUND I, Ø.5
3Ø NEXT
4Ø FOR I=1ØØØ TO 44Ø STEP -5
5Ø SOUND I,7 Ø.5
6Ø NEXT
```

SPACE$
Function

Purpose: Returns a string consisting of *n* spaces.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = SPACE$(*n*)

Remarks: *n* must be in the range Ø to 255.

See also "SPC Function."

Example: This example uses the SPACE$ function to print each number I on a line preceded by I spaces. An additional space is inserted because BASIC puts a space in front of positive numbers.

```
1Ø FOR I = 1 TO 5
2Ø X$ = SPACE$(I)
3Ø PRINT X$;I
4Ø NEXT I
RUN
 1
  2
   3
    4
     5
```

SPC
Function

Purpose: Skips *n* spaces in a PRINT statement.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: PRINT SPC(*n*)

Remarks: *n* must be in the range Ø to 255.

If *n* is greater than the defined width of the device, the value used is *n* MOD *width*. SPC can be used only with PRINT, LPRINT, and PRINT # statements.

If the SPC function is at the end of the list of data items, BASIC does not add a carriage return, as though the SPC function had an implied semicolon after it.

See also "SPACE$ Function."

Example: This example prints OVER and THERE separated by 15 spaces.

```
PRINT "OVER" SPC(15) "THERE"
OVER               THERE
```

SQR
Function

Purpose: Returns the square root of x.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: v = SQR(x)

Remarks: x must be greater than or equal to zero.

In BASIC 2.Ø and later releases, you can have this calculation performed in double-precision by specifying **/D** on the BASIC command line when BASIC is initially loaded. See "Options in the BASIC Command" in the *BASIC Handbook*.

Example: This example calculates the square roots of the numbers 1Ø, 15, 2Ø, and 25.

```
1Ø FOR X = 1Ø TO 25 STEP 5
2Ø PRINT X, SQR(X)
3Ø NEXT
RUN
 1Ø            3.162278
 15            3.872984
 2Ø            4.472136
 25            5
```

STICK
Function

Purpose: Returns the *x* and *y* coordinates of two joysticks.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = STICK(*n*)

Remarks:

n is a numeric expression in the range Ø to 3 that affects the result as follows:

0 returns the *x* coordinate for joystick A.

1 returns the *y* coordinate of joystick A.

2 returns the *x* coordinate of joystick B.

3 returns the *y* coordinate of joystick B.

Note: STICK(Ø) retrieves all four values for the coordinates, and returns the value for STICK(Ø). STICK(1), STICK(2), and STICK(3) get the values previously retrieved by STICK(Ø).

The range of values for *x* and *y* depends on your particular joysticks.

STICK
Function

Example: This program prints 1ØØ samples of the coordinates of joystick B.

```
1Ø PRINT "Joystick B"
2Ø PRINT "x coordinate","y coordinate"
3Ø FOR J=1 TO 1ØØ
4Ø TEMP=STICK(Ø)
5Ø X=STICK(2): Y=STICK(3)
6Ø PRINT X,Y
7Ø NEXT
```

Purpose: Stops execution of a program and returns to command level.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: STOP

Remarks: STOP statements can be used anywhere in a program to stop execution. When BASIC encounters a STOP statement, it displays the following message:

```
Break in nnnnn
```

at which nnnnn is the line number where the STOP occurred.

Unlike the END statement, the STOP statement does not close files.

BASIC always returns to command level after it executes a STOP. You can resume execution of the program by issuing a CONT command. See "CONT Command."

STOP
Statement

Example: This example calculates the value of TEMP, then stops. While the program is stopped, you can check the value of TEMP. Then you can use CONT to resume program execution at line 4Ø.

```
1Ø INPUT A, B
2Ø TEMP= A*B
3Ø STOP
4Ø FINAL = TEMP+2ØØ: PRINT FINAL
RUN
? 26, 2.1
Break in 3Ø
PRINT TEMP
 54.6
CONT
 254.6
```

STR$ Function

Purpose: Returns a string representation of the value of *x*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = STR$(*x*)

Remarks: *x* is any numeric expression.

If *x* is positive, the string returned by STR$ contains a leading blank (the space reserved for the plus sign). For example:

```
? STR$(321); LEN(STR$(321))
 321 4
```

The VAL function is complementary to STR$. See "VAL Function."

Example: This example branches to different sections of the program according to the number of digits in a number that is entered. The number of digits are counted by using STR$ to convert the number to a string; then the program branches, based on the length of the string.

```
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N))-1 GOSUB 30,100,200,300
    .
    .
    .
```

STRIG
Statement and Function

Purpose: Returns the status of the joystick buttons (triggers).

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: As a statement:

STRIG ON

STRIG OFF

As a function:

v = STRIG(n)

Remarks:

n is a numeric expression in the range Ø to 3. It affects the value returned by the function as follows:

Ø Returns -1 if button A1 was pressed since the last STRIG(Ø) function call; returns Ø if not.

1 Returns -1 if button A1 is currently pressed; returns Ø if not.

2 Returns -1 if button B1 was pressed since the last STRIG(2) function call; returns Ø if not.

3 Returns -1 if button B1 is currently pressed; returns Ø if not.

STRIG
Statement and Function

In Advanced BASIC and the BASIC Compiler, you can read four buttons from the joysticks. The additional values for *n* are:

4 Returns -1 if button A2 was pressed since the last STRIG(4) function call; returns Ø if not.

5 Returns -1 if button A2 is currently pressed; returns Ø if not.

6 Returns -1 if button B2 was pressed since the last STRIG(6) function call; returns Ø if not.

7 Returns -1 if button B2 is currently pressed; returns Ø if not.

STRIG ON must be executed before any STRIG(*n*) function calls can be made. After STRIG ON, every time the program starts a new statement, BASIC checks to see if a button was pressed.

If STRIG is OFF, no testing takes place.

See the next entry, "STRIG(*n*) Statement," for enhancements to the STRIG function in Advanced BASIC.

STRIG(n) Statement

Purpose: Enables and disables trapping of the joystick buttons.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format:

STRIG(*n*) ON
STRIG(*n*) OFF
STRIG(*n*) STOP

Remarks:

n can be Ø, 2, 4, or 6 and indicates the button to be trapped as follows:

0	button A1
2	button B1
4	button A2
6	button B2

STRIG(*n*) ON must be executed to enable trapping by the ON STRIG(*n*) statement. See "ON STRIG Statement." After STRIG(*n*) ON, every time the program starts a new statement, BASIC checks to see if the specified button has been pressed.

If STRIG(*n*) OFF is executed, no testing or trapping takes place. Even if the button is pressed, the event is not remembered.

If a STRIG(*n*) STOP statement is executed, no trapping takes place. However, if the button is pressed it is remembered so that an immediate trap takes place when STRIG(*n*) ON is executed.

Purpose: Returns a string length *n* whose characters all have ASCII code *m* or the first character of *x$*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = STRING$(*n,m*)

*v$* = STRING$(*n,x$*)

Remarks:

n, m are in the range Ø to 255.

x$ is any string expression.

Example: This example repeats an ASCII value of 45 to print a string of hyphens.

```
1Ø X$ = STRING$(1Ø,45)
2Ø PRINT X$ "MONTHLY REPORT" X$
RUN
----------MONTHLY REPORT----------
```

This example repeats the first character of the string "ABCD".

```
1Ø X$="ABCD"
2Ø Y$=STRING$(1Ø,X$)
3Ø PRINT Y$
RUN
AAAAAAAAAA
```

SWAP Statement

Purpose: Exchanges the values of two variables.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: SWAP *variable1, variable2*

Remarks:

variable1, variable2
are the names of two variables or array elements.

Any type variable can be swapped (integer, single-precision, double-precision, string), but the two variables must be of the same type or a **Type mismatch** error results.

Example: In this example, after line 3Ø is executed, A$ has the value " ALL " and B$ has the value " ONE ".

```
1Ø A$=" ONE " : B$=" ALL " : C$="FOR"
2Ø PRINT A$;C$;B$
3Ø SWAP A$, B$
4Ø PRINT A$;C$;B$
RUN
 ONE FOR ALL
 ALL FOR ONE
```

SYSTEM Command

Purpose: Exits BASIC and returns to DOS.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: SYSTEM

Remarks: SYSTEM closes all files before it returns to DOS. Your BASIC program is lost.

If you entered BASIC through a batch file from DOS, the SYSTEM command returns you to the batch file, which continues executing at the next statement.

TAB Function

Purpose: Tabs to position *n*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: PRINT TAB(*n*)

Remarks: *n* must be in the range 1 to 255.

If the current print position is already beyond space *n*, TAB goes to position *n* on the next line. Space 1 is the leftmost position, and the rightmost position is the defined WIDTH.

TAB can be used only in PRINT, LPRINT, and PRINT # statements.

If the TAB function is at the end of the list of data items, BASIC does not add a carriage return, as though the TAB function had an implied semicolon after it.

Example: TAB is used in the following example to cause the information on the screen to line up in columns.

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "L. M. JACOBS","$25.00"
RUN
NAME                      AMOUNT

L. M. JACOBS              $25.00
```

TAN
Function

Purpose: Returns the trigonometric tangent of *x*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = TAN(*x*)

Remarks:

x is the angle in radians. To convert degrees to radians, multiply by PI/18Ø, where PI=3.141593.

In BASIC 2.Ø and later releases, you can have this calculation performed in double-precision by specifying **/D** in the BASIC command line when BASIC is initially loaded. See "Options in the BASIC Command" in the *BASIC Handbook*.

Example: This example calculates the tangent of 45 degrees.

```
1Ø PI=3.141593
2Ø DEGREES=45
3Ø PRINT TAN(DEGREES*PI/18Ø)
RUN
 1
```

TIME$
Variable and Statement

Purpose: Sets or retrieves the current time.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: As a variable:

*v$* = TIME$

As a statement:

TIME$ = *x$*

Remarks: **For the variable (v$ = TIME$):**

The current time is returned as an 8-character string. The string is of the form *hh:mm:ss*, where *hh* is the hour (ØØ to 23), *mm* is the minutes (ØØ to 59), and *ss* is the seconds (ØØ to 59). (You may have set the time in DOS before you invoked BASIC.)

For the statement (TIME$ = x$):

The current time is set. *x$* is a string expression indicating the time to be set. *x$* can be given in one of the following forms:

hh	Set the hour in the range Ø to 23. Minutes and seconds default to ØØ.
hh:mm	Set the hour and minutes. Minutes must be in the range Ø to 59. Seconds default to ØØ.
hh:mm:ss	Set the hour, minutes, and seconds. Seconds must be in the range Ø to 59.

A leading zero can be omitted from any of the above values, but you must include at least one digit. For example, if you want to set the time as a half hour after midnight, you can enter

```
TIME$="Ø:3Ø"
```

but not

```
TIME$=":3Ø"
```

If any of the values are out of range, an **Illegal function call** error is issued. The previous time is retained. If *x$* is not a valid string, a **Type mismatch** error results.

Example: The following program continuously displays the time on the screen.

```
1Ø CLS
2Ø LOCATE 1Ø,15
3Ø PRINT TIME$
4Ø GOTO 2Ø
```

TIMER Function

Purpose: Returns a single-precision number representing the number of seconds elapsed since midnight or a system reset. (For BASIC 2.Ø and later releases.)

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	

Format: *v*=TIMER

Remarks: Fractional seconds are calculated to the nearest degree possible. TIMER is a read-only function.

Example: This example illustrates how TIMER resets after midnight. Values may be slightly different for your system.

```
1Ø TIME$="23:59:59"
2Ø FOR I=1 TO 2Ø
3Ø PRINT "TIME$= ";TIME$;" TIMER=";TIMER
4Ø NEXT
RUN

TIME$= 23:59:59 TIMER= 86399.Ø6
TIME$= 23:59:59 TIMER= 86399.11
TIME$= 23:59:59 TIMER= 86399.18
                  .
                  .
                  .
TIME$= 24:ØØ:ØØ TIMER= Ø
TIME$= ØØ:ØØ:ØØ TIMER= .Ø5
TIME$= ØØ:ØØ:ØØ TIMER= .16
TIME$= ØØ:ØØ:ØØ TIMER= .21
```

TRON and TROFF Commands

Purpose: Traces the execution of program statements.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: TRON

TROFF

Remarks: As an aid in debugging, the TRON command (which can be entered from within a program or in direct mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace is turned off by the TROFF command.

Example: This example uses TRON and TROFF to trace execution of a loop. The numbers in brackets are line numbers; the numbers not in brackets at the end of each line are the values of J, K, and L, which are printed by the program.

```
10 K=10
20 FOR I=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
TRON
RUN
[10][20][30][40] 0  10  20
[50][60][30][40] 0  20  30
[50][60][70]
TROFF
```

USR Function

Purpose: Calls the indicated assembly language subroutine with the argument *arg*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: *v* = USR[*n*](*arg*)

Remarks:

n is an integer in the range Ø-9 and corresponds to the digit supplied with the DEF USR statement for the desired routine. If *n* is omitted, USRØ is assumed. See "DEF USR Statement."

arg is any numeric expression or string variable that is the argument to the assembly language subroutine. If the subroutine does not require an argument, *arg* must still be supplied as a dummy argument.

The CALL statement is another way to interface with assembly language subroutines. See "CALL Statement." and also Appendix B, "Assembly Language Subroutines," for more information.

When the USR function is called, register AL contains a value that specifies the type of argument supplied. The value in AL will be one of the following:

Value in AL	Type of Argument
2	2-byte integer (twos complement)
3	String
4	Single-precision number
8	Double-precision number

If the argument is a string, the DX register points to the 3-byte string descriptor. See Appendix B, "Assembly Language Subroutines," under "How BASIC Interfaces with Assembly Language Subroutines" for information on the string descriptor.

If the argument is a number and not a string, the value of the argument is placed in the Floating Point Accumulator (FAC), which is an 8-byte area in BASIC's data space. In this case, the BX register contains the offset within the BASIC data space to the fifth byte of the 8-byte FAC. For the following examples, assume that the FAC is in bytes hex 49F through hex 4A6; that is, BX contains hex 4A3:

If the argument is an integer:

- Hex 4A4 contains the upper 8 bits of the argument.
- Hex 4A3 contains the lower 8 bits of the argument.

If the argument is a single-precision number:

- Hex 4A6 contains the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa. Hex 4A5 contains

the highest 7 bits of the mantissa with the leading 1 suppressed (implied). Bit 7 is the sign of the number (Ø = positive; 1 = negative).

- Hex 4A4 contains the middle 8 bits of the mantissa.

- Hex 4A3 contains the lowest 8 bits of the mantissa.

If the argument is a double-precision number:

- Hex 4A3 through hex 4A6 are the same as described under single-precision floating-point number in the preceding paragraph.

- Hex 49F through Hex 4A2 contain 4 more bytes of the mantissa (hex 49F contains the lowest 8 bits).

Usually, the value returned by a USR function is the same type (integer, string, single-precision, or double-precision) as the argument that was passed to it. However, a numerical argument of the function, regardless of its type, can be forced to an integer value by calling the FRCINT routine to get the integer equivalent of the argument placed into register BX.

If the value being returned by the function is to be an integer, place the resulting value into the BX register. Then make a call to MAKINT just before the intersegment return. This passes the value back to BASIC by placing it into the FAC.

Example: The methods for accessing FRCINT and MAKINT are shown in the following example:

```
10 DEFINT A-Z
20 OPTION BASE 1
30 X=0: Y=0: Z=0:
40 DIM ARRAY(&H14)
50 Z = VARPTR(ARRAY(1))
60 BLOAD"SUBRT.COM",Z
70 DEF USR0= Z
80 X=5
90 Y=USR0(X)
100 PRINT Y
```

The following assembly language subroutine has been loaded into an integer array. This subroutine doubles the argument passed and returns an integer result.

```
            PAGE,132        ;
            TITLE SUBRT.ASM
;                           BLOAD header
AAA         SEGMENT PARA PUBLIC 'CODE'
            DB      ØFDH
            DW      Ø,Ø
            DW      TRAILER-HEADER
AAA         ENDS
;
;
RSEG        SEGMENT AT ØF6ØØH ;base of BASIC ROM
            ORG   3       ;offset to force integer
FRCINT      LABEL FAR
            ORG   7       ;offset to make integer
MAKINT      LABEL FAR
RSEG        ENDS
;
;
CSEG        SEGMENT BYTE PUBLIC 'CODE'
HEADER      EQU    $
USRPRG      PROC  FAR     ;entry point
;call  FRCINT and force argument
;in FAC into [BX]
            DB    9AH
            DW    FRCINT ;offset
            DW    ØF6ØØH ;segment id
            ADD   BX,BX  ;[BX]=[BX]*2
;call MAKINT and put integer result
;in [BX] into FAC
            DB    9AH
            DW    MAKINT ;offset
            DW    ØF6ØØH ;segment id
            RET          ;intersegment return
USRPRG      ENDP
TRAILER     EQU   $
CSEG        ENDS
            END
```

Note: FRCINT and MAKINT perform intersegment returns. Make sure that the calls to FRCINT and MAKINT are defined by a FAR procedure.

VAL
Function

Purpose: Returns the numerical value of string *x$*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = VAL(*x$*)

Remarks: *x$* is a string expression.

The VAL function strips blanks, tabs, and line feeds from the argument string to determine the result. For example,

```
VAL("  -3")
```

returns -3.

If the first characters of *x$* are not numeric, VAL(*x$*) returns Ø (zero).

See "STR$ Function" for numeric to string conversion.

Example: In this example, VAL is used to extract the house number from an address.

```
PRINT VAL("34Ø8 SHERWOOD BLVD.")
 34Ø8
```

VARPTR Function

Purpose: Returns the offset to the current segment of memory of the variable.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = VARPTR(*variable*)

Remarks:

variable is the name of a numeric or string variable or array element in your program. A value must be assigned to *variable* before the call to VARPTR, or an **Illegal function call** error results.

VARPTR returns an offset in the range Ø to 65535. This number is the offset into BASIC's data segment of the first byte of data identified with *variable*. The format of this data is described in Appendix B, Memory Information, in the *BASIC Handbook*, under "How Variables Are Stored."

> **Note:** All simple variables should be assigned before calling VARPTR for an array, because arrays will be relocated whenever a new simple variable is assigned.

VARPTR is usually used to get the offset of a variable or array to BASIC's data segment so it can be used to access assembly language subroutines and pass variables or arrays to them.

VARPTR
Function

Example: This example uses VARPTR to get the data from a variable. In line 3Ø, P gets the address of the data. Integer data is stored in two bytes, with the less significant byte first. The actual value stored at location P is calculated in line 4Ø. The bytes are read with the PEEK function, and the second byte is multiplied by 256 because it contains the high-order bits.

```
1Ø DEFINT A-Z
2Ø DATA1=5ØØ
3Ø P=VARPTR(DATA1)
4Ø V=PEEK(P) + 256*PEEK(P+1)
5Ø PRINT V
```

VARPTR$ Function

Purpose: Returns a character form of the offset of a variable in memory. It is primarily for use with PLAY and DRAW in programs that will be compiled.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: *v$* = VARPTR$(*variable*)

Remarks:

variable is the name of a variable in the program.

Note: All simple variables should be assigned before calling VARPTR$ for an array element, because arrays are relocated whenever a new simple variable is assigned.

VARPTR$ returns a 3-byte string in the form:

Byte Ø	**Byte 1**	**Byte 2**
type	low byte of variable address	high byte of variable address

type indicates the variable type:

2 integer
3 string
4 single-precision
8 double-precision

VARPTR$
Function

The returned value is the same as:

```
CHR$(type)+MKI$(VARPTR(variable))
```

You can use VARPTR$ to indicate a variable name in the command string for PLAY or DRAW. For example:

Method One	Alternative Method
PLAY "XA$;"	PLAY "X"+VARPTR$(A$)
PLAY "O=I;"	PLAY "O="+VARPTR$(I)

Purpose: Defines a rectangular subset of the screen onto which WINDOW and WINDOW contents are mapped. (For BASIC 2.Ø and later releases.)

Versions: Cassette Disk Advanced *** Compiler

Graphics mode only.

Format: VIEW [[*SCREEN*] [(*x1,y1*)- (*x2,y2*) [,[*color*] [,[*boundary*]]]]]

Remarks:

(x1,y1)-(x2,y2)
are the upper-left (*x1,y1*) and the lower-right (*x2,y2*) coordinates of the viewport defined. The *x* and *y* coordinates must be within the actual limits of the screen or an **Illegal function call** error occurs. For more information, see "Specifying Coordinates" under "Graphics Modes" in Chapter 3 of the *BASIC Handbook*.

color
lets you fill the defined viewport with color. If *color* is omitted, the viewport is not filled. *color* is an integer expression that chooses an attribute from the attribute range for the current screen mode. In medium resolution, the color is the current one for that attribute as defined by the COLOR statement. Four attributes (Ø-3)

are available in medium resolution; in high resolution, two attributes (Ø-1) are available. Zero (Ø) is always the attribute for the background. The default foreground attribute is always the maximum attribute for that screen mode: 3 in medium resolution; 1 in high resolution.

boundary lets you draw a boundary line around the viewport (if space is available). If *boundary* is omitted, no boundary is drawn. *boundary* is an integer expression in the range described in *color*.

It is important to note that VIEW sorts the *x* and *y* argument pairs, placing the smaller values for *x* and *y* first. For example:

```
VIEW (1ØØ,1ØØ)-(5,5)
```

becomes:

```
VIEW (5,5)-(1ØØ,1ØØ)
```

Another example:

```
VIEW (31Ø,1ØØ)-(2ØØ,15Ø)
```

becomes:

```
VIEW (2ØØ,1ØØ)-(31Ø,15Ø)
```

All possible pairings of *x* and *y* are valid. The only restriction is that *x1* cannot equal *x2* and *y1* cannot equal *y2*. The viewport cannot be larger than the viewing surface.

If the SCREEN argument is omitted, all points plotted are relative to the viewport. That is, *x1* and *y1* are added to the *x* and *y* coordinates before plotting the point on the screen. For example, if:

```
10 VIEW (10,10)-(200,100)
```

is executed, the point plotted by PSET (Ø,Ø),3 is at the actual screen location 1Ø,1Ø.

If the SCREEN argument is included, all points plotted are absolute and can be inside or outside the screen limits. However, only those points within the viewport limits are visible. For example if:

```
10 VIEW SCREEN (10,10)-(200,100)
```

is executed, the point plotted by PSET (Ø,Ø),3 does not appear on the screen because Ø,Ø is outside the viewport. PSET (1Ø,1Ø),3 is within the viewport and places the point in the upper-left corner.

VIEW with no arguments defines the entire screen as the viewport.

You can define multiple viewports, but only one viewport can be active at a time. RUN and change in SCREEN attributes disable the viewports.

VIEW used with WINDOW allows you to scale images. See the second example. See also "WINDOW Statement."

> **Note:** When VIEW is used, the CLS statement clears only the current viewport. To clear the entire screen, you must use VIEW to disable the viewports, and then use CLS to clear the screen.

With viewports, CLS does not move the cursor to the home position. Use Ctrl-Home to send the cursor home and clear the screen.

Examples: The following example defines four viewports:

```
10 SCREEN 1:VIEW:CLS:KEY OFF
20 VIEW (1,1)-(151,91),,1
30 VIEW (165,1)-(315,91),,2
40 VIEW (1,105)-(151,195),,2
50 VIEW (165,105)-(315,195),,1
60 LOCATE 2,4:PRINT "Viewport 1"
70 LOCATE 2,25:PRINT "Viewport 2"
80 LOCATE 15,4:PRINT "Viewport 3"
90 LOCATE 15,25:PRINT "Viewport 4"
100 VIEW (1,1)-(151,91):GOSUB 1000
200 VIEW (165,1)-(315,91):GOSUB 2000
300 VIEW (1,105)-(151,195):GOSUB 3000
400 VIEW (165,105)-(315,195):GOSUB 4000
900 END
1000 CIRCLE (65,50),30,2
1010 'Draw a circle in first viewport
1020 RETURN
2000 LINE (45,50)-(90,75),1,B
2010 'Draw a box in second viewport
2020 RETURN
3000 FOR D=0 TO 360:DRAW "ta=d;nu20":NEXT
3010 'Draw spokes in third viewport
3020 RETURN
4000 PSET(60,50),2:DRAW "e15;f15;130"
4010 'Draw a triangle in fourth viewport
4020 RETURN
```

This example demonstrates scaling with VIEW.

```
10 KEY OFF:CLS:SCREEN 1,0:COLOR 0,0
20 WINDOW SCREEN(320,0)-(0,200)
30 GOTO 80
40 C=1
50 CIRCLE (160,100),60,C,,,5/18
60 CIRCLE (160,100),60,C,,,1
70 RETURN
80 GOSUB 460:FOR I=1 TO 1500:NEXT I: CLS
90 VIEW (1,1)-(160,90),,2:GOSUB 40
100 GOTO 100
```

WAIT
Statement

Purpose: Suspends program execution while monitoring the status of a machine input port.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: WAIT *port, n*[,*m*]

Remarks:

port is the port number, in the range Ø to 65535.

n, m are integer expressions in the range Ø to 255.

See the IBM Personal Computer *Technical Reference* for a description of valid port numbers (I/O addresses).

The WAIT statement suspends program execution until a specified machine input port develops a specified bit pattern.

The data read at the port is XORed with the integer expression *m* and then ANDed with *n*. If the result is zero, BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If *m* is omitted, it is assumed to be zero.

The WAIT statement lets you test one or more bit positions on an input port. You can test the bit position for either a 1 or a Ø. The bit positions to be tested are specified by setting 1's in those positions in *n*. If you do not specify *m*, the input port bits are

tested for 1's. If you specify *m*, a 1 in any bit position in *m* (for which there is a 1 bit in *n*) causes WAIT to test for a Ø for that input bit.

When executed, the WAIT statement loops, testing those input bits specified by 1's in *n*. If *any one* of those bits is 1 (or Ø if the corresponding bit in *m* is 1), then the program continues with the next statement. Thus WAIT does not wait for an entire pattern of bits to appear, but only for one of them to occur.

> **Note:** It is possible to enter an infinite loop with the WAIT statement. You can do a Ctrl-Break or a System Reset to exit the loop.

Example: To suspend program execution until port 32 receives a 1 bit in the second bit position:

```
1ØØ WAIT 32,2
```

WHILE and WEND Statements

Purpose: Executes a series of statements in a loop as long as a given condition is true.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format:

```
WHILE expression
.
.
.
(loop statements)
.
.
.
WEND
```

Remarks:

expression is any numeric expression.

If *expression* is true (not zero), *loop statements* execute until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks *expression*. If expression is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE–WEND loops can be nested to any level. Each WEND matches the most recent WHILE. An unmatched WHILE statement causes a **WHILE without WEND** error, and an unmatched WEND statement causes a **WEND without WHILE** error.

Example: The following example sorts the elements of the array A into alphabetical order. A was defined with J elements.

```
10 'bubble sort array A
20 FLIPS=1 'force first pass thru loop
30 WHILE FLIPS
40 FLIPS=0
50 FOR I=1 TO J-1
60 IF A(I)>A(I+1) THEN
        SWAP A(I),A(I+1): FLIPS=1
70 NEXT I
80 WEND
```

WIDTH
Statement

Purpose: Sets the output line width in number of characters. After outputting the indicated number of characters, BASIC adds a carriage return.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: WIDTH *size*

WIDTH *device,size*

WIDTH #*filenum,size*

Remarks:

size is a numeric expression in the range Ø to 255. This is the new width. WIDTH Ø is the same as WIDTH 1.

device is a string expression for the device identifier. Valid devices are SCRN:, LPT1:, LPT2:, LPT3:, COM1:, or COM2:.

filenum is a numeric expression in the range 1 to 15. This is the number of a file opened to an output device.

Depending on the device specified, the following actions are possible:

WIDTH size or WIDTH "SCRN:" size
Sets the screen width. Only 4Ø- or 8Ø-column widths are allowed. WIDTH 4Ø is not valid for the IBM Monochrome Display.

If the screen is in medium-resolution graphics mode (as occurs with a SCREEN 1 statement), WIDTH 8Ø forces the screen into high resolution (as with a SCREEN 2 statement). The reverse if true when in high resolution.

Note: Changing the screen width causes the screen to be cleared, and sets the border screen color to black.

WIDTH device,size

Is a deferred width assignment for the device. This form of width stores the new width value without changing the current width setting. A subsequent OPEN to the device will use this value for width while the file is open. The width does not change immediately if the device is already open.

Note: LPRINT, LLIST, and LIST,"LPTn" do an implicit OPEN and are therefore affected by this statement.

WIDTH #filenum,size

The width of the device associated with *filenum* is immediately changed to the new size specified. This allows the width to be changed at will while the file is open. This form of WIDTH has meaning only for LPT1: in Cassette BASIC. Disk BASIC and Advanced BASIC also allow LPT2:, LPT3:, COM1:, and COM2:. Note that the number sign (#) is required.

Any value entered outside the ranges indicated results in an **Illegal function call** error. The previous value is retained.

WIDTH has no effect for the keyboard (KYBD:) or cassette (CAS1:).

The width for each printer defaults to 8Ø when BASIC is started. The maximum width for the IBM 8Ø CPS Matrix Printer is 132. However, no error is returned for values between 132 and 255.

It is up to you to set the appropriate physical width on your printer. Some printers are set by sending special codes; some have switches. For the IBM 8Ø CPS Matrix Printer you should use LPRINT CHR$(15); to change to a condensed typestyle when printing at widths greater than 8Ø. Use LPRINT CHR$(18); to return to normal. The IBM 8Ø CPS Matrix Printer is set up to automatically add a carriage return if you exceed the maximum line length.

Specifying a width of 255 disables line folding. This has the effect of "infinite" width. WIDTH 255 is the default for communications files.

Changing the width for a communications file does not change either the receive or the transmit buffer; it just causes BASIC to send a carriage return character after every *size* character.

Changing screen mode affects screen width only when moving between SCREEN 2 and SCREEN 1 or SCREEN Ø. See "SCREEN Statement."

Example: In this example, line 1Ø stores a printer width of 75 characters per line. Line 2Ø opens file #1 to the printer and sets the width to 75 for subsequent PRINT #1,... statements. Line 6Ø2Ø changes the current printer width to 4Ø characters per line. Notice that the WIDTH value must come before the OPEN statement.

```
1Ø WIDTH "LPT1:",75
2Ø OPEN "LPT1:" FOR OUTPUT AS #1
   .
   .
   .
6Ø2Ø WIDTH #1,4Ø
```

These examples change screen mode and width.

```
SCREEN 1,Ø  'Set to med-res color graphics
WIDTH 8Ø    'Go to hi-res graphics
WIDTH 4Ø    'Go back to medium res
SCREEN Ø,1  'Go to 4Øx25 text color mode
WIDTH 8Ø    'Go to 8Øx25 text color mode
```

WINDOW Statement

Purpose: Redefines the coordinates of the viewport. (For BASIC 2.Ø and later releases.)

Versions: Cassette Disk Advanced *** Compiler

Graphics mode only.

Format: WINDOW [[SCREEN] (*x1,y1*)- (*x2,y2*)]

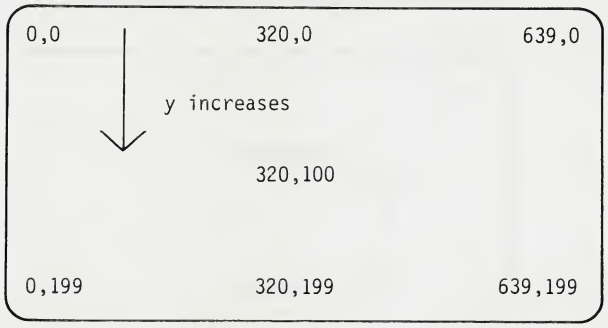
(x1,y1),(x2,y2)
are programmer-defined coordinates called *world coordinates*. These coordinates are single-precision, floating-point numbers. They define the world coordinate space that will be mapped into the the physical coordinate space, as defined by the VIEW statement. See "VIEW Statement."

WINDOW allows you to draw objects in space ("world coordinate system") and not be bounded by the limits of the screen ("physical coordinate system"). This is done by specifying the world coordinate pairs (*x1,y1*) and (*x2,y2*). BASIC then converts world coordinate pairs for subsequent display within the viewport. To make this transformation from world space to the physical space of the screen, BASIC must know what portion of the unbounded world coordinate space contains the information you want to be displayed. This rectangular region in the world coordinate space is called a *window*.

In the physical coordinate system, if you run the following:

```
NEW
SCREEN 2
```

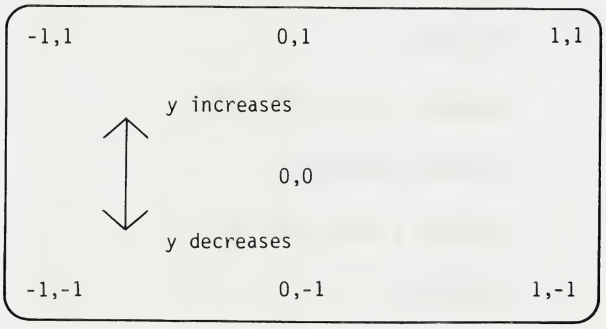
the screen appears with standard coordinates as:



When the SCREEN attribute is omitted, the screen is viewed in true Cartesian coordinates. For example, given:

```
WINDOW (-1,-1)-(1,1)
```

the screen appears as:



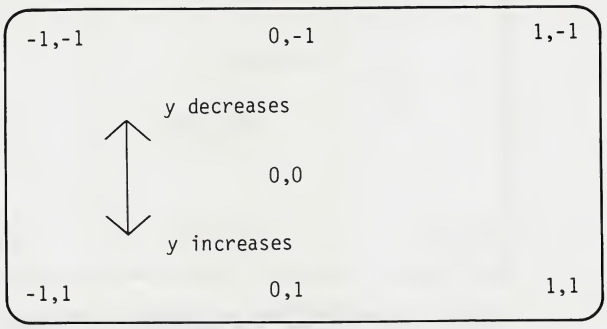
Note that the *y* coordinate is inverted so that *(x1,y1)* is the lower-left coordinate and *(x2,y2)* is the upper-right coordinate.

WINDOW
Statement

When the SCREEN attribute is included, the coordinates are not inverted so that *(x1,y1)* is the upper-left coordinate and *(x2,y2)* is the lower-right coordinate. For example:

```
WINDOW SCREEN (-1,-1)-(1,1)
```

defines the screen to look like this:



It is important to note that WINDOW sorts the *x* and *y* argument pairs, placing the smaller values for *x* and *y* first. For example:

```
WINDOW (1ØØ,1ØØ)-(5,5)
```

becomes:

```
WINDOW (5,5)-(1ØØ,1ØØ)
```

Another example:

```
WINDOW (-4,4)-(4,-4)
```

becomes:

```
WINDOW (-4,-4)-(4,4)
```

All possible pairings of *x* and *y* are valid. The only restriction is that *x1* cannot equal *x2* and *y1* cannot equal *y2*.

WINDOW also allows you to "zoom" and "pan." Using a window with coordinates larger than an image displays the entire image, but the image is small and blank spaces appear on the sides of the screen. Choosing window coordinates smaller than an image forces clipping and allows only a portion of the image to be displayed and magnified. By specifying small and large window sizes, you can zoom in until an object occupies the entire screen, or you can zoom out until the image is just a spot on the screen.

RUN, SCREEN, and WINDOW with no attributes disable any WINDOW definitions and return the screen to physical coordinates.

Examples: The following example shows clipping using WINDOW.

```
10 SCREEN 2:CLS
20 WINDOW (-6,-6)-(6,6)
30 CIRCLE (4,4),5,1
40 'the circle is large and only part is visible
50 WINDOW (-100,-100)-(100,100)
60 CIRCLE (4,4),5,1 'the circle is very small
70 END
```

WINDOW
Statement

The following example shows the effect of zooming using WINDOW.

```
10 KEY OFF:CLS:SCREEN 1,0
20 '
30 GOTO 160
40 '====================
50 'procedure display
60 '
70 LINE (X,0)-(-X.0),,,&HAA00 'create x axis
80 LINE (0,X)-(0,-X),,,&HAA00 'create y axis
90 '
100 CIRCLE (X/2,X/2),R 'circle has radius r
110 FOR P=1 TO 50:NEXT P 'delay loop
120 '
130 RETURN
140 '===================
150 '
160 X=1000:WINDOW (-X,-X)-(X,X):R=20
170 'create a graph with large coord range
180 GOSUB 50:FOR P=1 TO 1000:NEXT P:CLS
190 '
200 X=60:WINDOW (-X,-X)-(X,X):R=20
210 'smaller coord range increase circle size
220 GOSUB 50:FOR P=1 TO 1000:NEXT P:CLS
230 '
240 X=100:WINDOW (-5,-5)-(X,X):R=20
250 'modify window to show only portion of axes
260 GOSUB 50:FOR P=1 TO 1000:NEXT P:CLS
270 '
280 PRINT ".... an example":PRINT".    of zooming.."
290 FOR P=1 TO 1500:NEXT P
300 CLS:T=-50:U=100:X=U
310 FOR I=1 TO 45
320     T=T + 1:U=U - 1:X=X-1:R=20
330     WINDOW (T,T)-(U,U):CLS:GOSUB 50
340 NEXT I
350 END
```

Purpose: Outputs data to the screen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: WRITE [*list of expressions*]

Remarks:

list of expressions
is a list of numeric and/or string expressions, separated by commas or semicolons.

If the list of expressions is omitted, a blank line is displayed. If the list of expressions is included, the values of the expressions are displayed on the screen.

When the values of the expressions are displayed, each item is separated from the one before it by a comma. Strings are delimited by quotation marks. After the last item in the list is printed, BASIC adds a carriage return/line feed.

WRITE is similar to PRINT. The difference between WRITE and PRINT is that WRITE inserts commas between the items as they are displayed and delimits strings with quotation marks. Also, positive numbers are not preceded by blanks.

WRITE
Statement

Example: The following example shows how WRITE displays numeric and string values.

```
10 A=80: B=90: C$=".THAT'S ALL"
20 WRITE A,B,C$
RUN
80,90,".THAT'S ALL"
```

WRITE # Statement

Purpose: Writes data to a sequential file.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: WRITE #*filenum, list of expressions*

Remarks:

filenum is the number under which the file was opened for output.

list of expressions is a list of string and/or numeric expressions, separated by commas or semicolons.

The difference between WRITE # and PRINT # is that WRITE # inserts commas between the items as they are written and delimits strings with quotation marks. Therefore, it is not necessary for you to put explicit delimiters in the list. Also, WRITE # does not put a blank in front of a positive number. A carriage return/line feed sequence is inserted after the last item in the list is written.

Example: Let A$=".CAMERA" and B$=".936Ø4-1". The statement:

```
WRITE #1,A$,B$
```

writes the following image to the file.

```
".CAMERA",".936Ø4-1"
```

A subsequent INPUT # statement:

```
INPUT #1,A$,B$
```

inputs ".CAMERA" to A$ and ".936Ø4-1" to B$.

Appendixes

Contents

Appendix A. Error Messages

If BASIC detects an error that causes a program to stop running, an error message is displayed. You can trap and test errors in a BASIC program using the ON ERROR statement and the ERR and ERL variables. For complete explanations of ON ERROR, ERR, ERL, ERDEV, and ERDEV$, see those entries in this manual.

This appendix lists alphabetically all BASIC error messages with an explanation of each message, as well as some suggestions for recovering from errors. The separate BASIC Quick Reference lists all messages in order by their associated numbers.

Number	Message
73	**Advanced feature** Your program specified an Advanced BASIC feature while you are using Disk BASIC. Start Advanced BASIC and rerun your program.
54	**Bad file mode** You tried to use PUT or GET with a sequential or a closed file; or to execute an OPEN with a file mode other than input, output, append, or random. Make sure the OPEN statement was entered and executed properly. GET and PUT require a random file. This error also occurs if you try to merge a file that is not in ASCII format. In this case,

make sure you are merging the right file. If necessary, load the program and save it again, using the **A** option.

64 **Bad file name**
An invalid form was used for the filename with KILL, NAME, or FILES.

Check "Naming Files" in Chapter 3 of the *BASIC Handbook* for information on valid filenames, and correct the filename in error.

52 **Bad file number**
A statement specified a file number of a file that is not open, or the file number is out of the range of possible file numbers specified at initialization. Or, the device name in the file specification is too long or invalid, or the filename is too long or invalid.

Make sure the file you wanted was opened and entered correctly in the statement. Check to see that you have a valid file specification. See "Naming Files" in Chapter 3 of the *BASIC Handbook* for information on file specifications.

63 **Bad record number**
In a PUT or GET (file) statement, the record number is either equal to zero or greater than the maximum allowed (32767 in BASIC versions earlier than 2.0). GET and PUT have subsequently been enhanced to allow record numbers in the range 1 to 16,777,215 to accommodate large files with short record numbers.

Correct the PUT or GET statement to use a valid record number.

17 **Can't continue**
You tried to use CONT to continue a program that:

- Halted because of an error
- Was changed during a break in execution
- Does not exist

Make sure the program is loaded, and use RUN to run it.

— **Can't continue after SHELL**
You shelled to a child process that terminated and remained resident. A child process cannot remain resident because it prevents BASIC from recovering its workspace, causing the program to halt.

69 **Communication buffer overflow**
A communication input statement was executed, but the input buffer was already full.

Use an ON ERROR statement to retry the input when this condition occurs. Subsequent inputs try to clear this fault unless characters continue to be received faster than the program can process them. If this happens there are several possible solutions:

- Increase the size of the communications buffer using the **/C:** option when you start BASIC.
- Implement a "hand-shaking" protocol with the other computer to tell it to stop sending long enough for you to catch up. See the example in Appendix C, "Communications."
- Use a lower baud rate to transmit and receive.

25 **Device fault**
A hardware error indication was returned by an interface adapter. In Cassette BASIC, this occurs only when a fault status is returned from the printer interface adapter.

This message can also occur when data is transmitted to a communications file. In this case, it indicates that one or more of the signals being tested (specified on the OPEN "COM... statement) were not found in the specified period of time.

57 **Device I/O error**
An error occurred on a device I/O operation. DOS cannot recover from the error.

When receiving communications data, this error can occur from overrun, framing, break, or parity errors. For data with 7 or less bits, the eighth bit is turned on in the byte in error.

24 **Device timeout**
BASIC did not receive information from an input/output device within a predetermined amount of time. In Cassette BASIC, this occurs only while the program is trying to read from the cassette or write to the printer.

For communications files, this message indicates that one or more of the signals tested with OPEN "COM... was not found in the specified period of time.

Retry the operation.

68 **Device unavailable**
You tried to open a file to a device that is not installed. Either you do not have the hardware to support the device (such as printer adapters for a second or third printer), or you have disabled the device.

(For example, you may have used **/C:0** in the BASIC command line to start Disk BASIC. That would disable communications devices.)

Make sure the device is installed correctly. If necessary, Return to DOS where you can reenter the BASIC command line.

66 **Direct statement in file**
A direct statement was encountered during a loading or chaining operation to an ASCII format file. The LOAD or CHAIN is terminated.

The ASCII file should consist only of statements preceded by line numbers. This error may occur because of a line feed character in the input stream.

61 **Disk full**
All disk storage space is in use. Files are closed when this error occurs.

If there are files on the disk that you no longer need, erase them or use a new disk. Then retry the operation or rerun the program.

72 **Disk media error**
The controller attachment card detected a hardware or media fault. Usually this means that the disk has gone bad.

Copy any existing files to a new disk and reformat the bad disk. If there are problems during reformatting, the disk should be discarded.

71 **Disk not ready**
The disk drive door is open or a disk is not in the drive.

Place the correct disk in the drive and continue the program.

70 **Disk write protect**
You tried to write to a disk that is write-protected.

Make sure you are using the right disk. If so, remove the write-protect tab; then retry the operation.

This error can also occur because of a hardware failure.

11 **Division by zero**
In an expression, you tried to divide by zero, or you tried to raise zero to a negative power.

It is not necessary to correct this condition, because the program continues running. Machine infinity with the sign of the number being divided is the result of the division; or, positive machine infinity is the result of the exponentiation. This error cannot be trapped using ON ERROR.

10 **Duplicate definition**
You tried to define the size of the same array twice. This can happen in one of several ways:

- The same array is defined in two DIM statements.
- The program encounters a DIM statement for an array after the default dimension of 10 is established for that array.

- The program sees an OPTION BASE statement after an array has been dimensioned, either by a DIM statement or by default.

Move the OPTION BASE statement to make sure it is executed before you use any arrays; or, fix the program so each array is defined only once.

50 **Field overflow**
A FIELD statement tried to allocate more bytes for the record length of a random file than were specified in the OPEN statement. Or, the end of the FIELD buffer was encountered during sequential I/O (PRINT #, WRITE #, INPUT #) to a random file.

Check the OPEN statement and the FIELD statement to make sure they correspond. If you are doing sequential I/O to a random file, make sure that the length of the data read or written does not exceed the record length of the random file.

58 **File already exists**
The filename specified in a NAME command duplicates an existing filename on the disk.

Retry the NAME command using a different name.

55 **File already open**
You tried to open a file for sequential output or append, and the file is already open; or, you tried to use KILL on a file that is open.

Make sure you execute only one OPEN to a file if you are writing to it sequentially. Close a file before you use KILL.

53 **File not found**
A LOAD, KILL, NAME, FILES, or OPEN references a file that does not exist on the disk in the specified drive.

Verify that the correct disk is in the drive specified, and that the file specification was entered correctly. Then retry the operation.

26 **FOR without NEXT**
A FOR was encountered without a matching NEXT. That is, a FOR loop was active when the physical end of the program was reached.

12 **Illegal direct**
You tried to enter a statement in direct mode that is invalid in that mode (such as DEF FN).

The statement should be entered as part of a program line.

5 **Illegal function call**
A parameter that is out of range was passed to a system function. The error can also occur as the result of:

- Using a negative or an unreasonably large subscript
- Trying to raise a negative number to a power that is not an integer
- Calling a USR function before defining the starting address with DEF USR
- Using a negative record number on GET or PUT (file)

- Using an improper argument to a function or statement
- Trying to list or edit a protected BASIC program
- Trying to delete line numbers that don't exist

See "BASIC Commands, Statements, and Functions" in this manual for information about the particular statement or function.

— **Incorrect DOS version**
The command you just entered requires a different version of DOS from the one you are running.

62 **Input past end**
This is an end-of-file error. An input statement was executed for a null (empty) file, or it was executed after all the data in a sequential file was already input.

To avoid this error, use the EOF function to detect the end of a file.

This error also occurs if you try to read from a file that was opened for output or append. If you want to read from a sequential output (or append) file, you must close it and open it again for input.

51 **Internal error**
An internal malfunction occurred in BASIC.

Recopy your disk. Check the hardware, following instructions in the *Guide to Operations* and retry the operation. If the error recurs, note the conditions under which the message appeared and notify the place of purchase.

23 **Line buffer overflow**
You tried to enter a line with too many characters.

If there are several statements on the line, move some of them to the next line. Also, use string variables instead of constants where possible.

22 **Missing operand**
An expression contains an operator, such as * or OR, with no operand following it.

Make sure you include all the required operands in the expression.

1 **NEXT without FOR**
The NEXT statement doesn't have a corresponding FOR statement. It may be that a variable in the NEXT statement does not correspond to any previously executed and unmatched FOR statement variable.

19 **No RESUME**
The program branched to an active error-trapping routine as a result of an error condition or an ERROR statement. The routine does not have a RESUME statement. (The physical end of the program was encountered in the error trapping routine.)

Be sure to include RESUME in your error-trapping routine to continue program execution. You may want to add an ON ERROR GOTO 0 statement to your error-trapping routine so BASIC displays the message for any untrapped error.

4 **Out of data**
A READ statement tried to read more data than is in the DATA statements.

7 **Out of memory**
A program is too large, has too many FOR loops or GOSUBs, too many variables, expressions that are too complicated, or complex painting.

You can use CLEAR at the beginning of your program to set aside more stack space or memory area.

27 **Out of paper**
The printer is out of paper or is not switched on.

Make sure that the power is on, verify that the printer is properly connected, insert paper if necessary; then, continue the program.

14 **Out of string space**
BASIC allocates string space dynamically until it runs out of memory. This message means that string variables caused BASIC to exceed the amount of free memory remaining after housecleaning.

6 **Overflow**
A number is too large to be represented in BASIC's number format. Integer overflow causes execution to stop. Otherwise, machine infinity with the appropriate sign is supplied as the result, and execution continues. Integer overflow is the only type of overflow that can be trapped.

To correct integer overflow, use smaller numbers, or change to single- or double-precision variables.

Note: If a number is too small to be represented in BASIC's number format, there is an *underflow* condition. If this occurs, the result is zero and execution continues without an error.

75 **Path/file access error**
During an OPEN, RENAME, MKDIR, CHDIR, or RMDIR operation, you tried to use a path or filename to an inaccessible file. For example, you tried to open a directory or volume identifier; you tried to open a read only file for writing; or you tried to remove the current directory. The operation is not completed.

76 **Path not found**
During an OPEN, MKDIR, CHDIR, or RMDIR operation, DOS was unable to find the path specified. The operation is not completed.

74 **Rename across disks**
You tried to rename a file, but you specified the wrong disk.

The rename operation is not performed.

When you use RENAME, the drive you specify must be the same for the old filename and the new filename. The exception to this is when the DOS ASSIGN command is active. The drive name can be different, but the same physical device must be specified.

20 **RESUME without error**
The program encountered a RESUME statement without having trapped an error. The error-trapping routine should be entered only when an error occurs or an ERROR statement is executed.

You probably need to include a STOP or END statement before the error-trapping routine to prevent the program from "falling into" the error-trapping code.

3 **RETURN without GOSUB**
A RETURN statement does not have a previous unmatched GOSUB statement.

Correct the program. You probably need to put a STOP or END statement before the subroutine so the program doesn't "fall into" the subroutine code.

16 **String formula too complex**
A string expression is too long or too complex. Break it into shorter expressions.

15 **String too long**
You tried to create a string more than 255 characters long. Break it into shorter strings.

9 **Subscript out of range**
You used an array element either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.

Check the usage of the array variable. You may have put a subscript on a variable that is not an array, or you may have coded a built-in function incorrectly.

2 **Syntax error**
A line contains an incorrect sequence of characters, such as an unmatched parenthesis, a misspelled command or statement, or incorrect punctuation.

Or, the data in a DATA statement doesn't match the type (numeric or string) of the variable in a READ statement.

Or, a reserved word has been used as a variable name. See Chapter 3 of the *BASIC Handbook* for a list of reserved words.

When this error occurs, the BASIC Program Editor automatically displays the line in error. Correct the line or the program.

67 **Too many files**
You tried to create a new file (using SAVE or OPEN) when all directory entries on the disk are full, or when the file specification is invalid.

If the file specification is correct, use a new formatted disk and retry the operation.

13 **Type mismatch**
You gave a string value where a numeric value was expected, or you had a numeric value in place of a string value. This error may also be caused by trying to SWAP variables of different types, such as single- and double-precision.

8 **Undefined line number**
A line that doesn't exist in the program was referred to in a command or statement.

Check the line numbers in your program, and use the correct line number.

18 **Undefined user function**
You called a function before defining it with the DEF FN statement.

Make sure the program executes the DEF FN statement before you use the function.

— **Unprintable error**
A specific error message is not available for the error condition that exists. This is usually caused by an ERROR statement with an undefined error code.

Check your program to make sure you handle all error codes.

30 **WEND without WHILE**
A WEND was encountered before a matching WHILE was executed.

Correct the program so that there is a WHILE for each WEND.

29 **WHILE without WEND**
A WHILE statement does not have a matching WEND. That is, a WHILE was still active when the physical end of the program was reached.

Correct the program so that each WHILE has a corresponding WEND.

— **You cannot SHELL to Basic**
BASIC cannot be run as a child process.

Appendix B. Assembly Language Subroutines

This appendix describes how BASIC interfaces with assembly language subroutines. In particular, it describes:

- Where in memory to locate your assembly language subroutines
- How BASIC interfaces with assembly language subroutines
- How to load and call your assembly language subroutines

This appendix is intended to be used by an experienced assembly language programmer.

Reference Material

The following publications contain related material that you may find useful.

Rector, Russell and Alexy, George. *The 8Ø86 Book*. Osborne/McGraw-Hill, Berkeley, California, 198Ø, (includes the 8Ø88)

Intel Corporation Literature Department. *The 8Ø86 Family User's Manual*, 98ØØ722. 3Ø65 Bowers Avenue, Santa Clara, CA 95Ø51.

IBM Corporation Personal Computer library. *Macro Assembler*. Boca Raton, FL 33432.

IBM Corporation Personal Computer library. *Technical Reference*. Boca Raton, FL 33432.

Deciding Where In Memory To Load Your Subroutines

BASIC normally uses all memory available from the starting location of its data area up to a maximum of 64K bytes. This area contains your BASIC program and data, along with the interpreter work area and BASIC's stack. You can allocate memory space for assembly language subroutines either inside or outside this BASIC 64K data segment. Where you put your assembly language subroutine depends on three items:

- Total amount of memory in your computer
- Size of your BASIC program
- Characteristics of your assembly language program

There are many methods for installing assembly language subroutines. We recommend these three places for storing them:

- As a resident part of DOS that is loaded before BASIC is invoked
- In an integer array within BASIC's data segment
- Beyond BASIC's data segment

DOS-Loaded Subroutines for BASIC

This is the recommended method for interfacing an assembly language subroutine from BASIC because it insures compatibility of your programs with DOS and BASIC now and in the future. It also reduces the margin for error when interfacing your subroutines. It does require additional planning in coding your subroutines. A sample subroutine later in this chapter shows you how this is done.

Features

- This method allows for multisegment subroutines, (up to 64K) and for nonself-relocating subroutines that would be impossible for BLOAD or DATA statements to handle properly.

- It safely makes the subroutine a resident part of DOS with no need to "hide" the subroutine from BASIC nor to reserve or protect space for the subroutine from BASIC or DOS.

Considerations

- DOS-loaded subroutines remain resident once they are loaded. Only rebooting the system removes the subroutine. In addition to this, multiple copies of the subroutine can become resident if it is invoked several times. This is best handled by defining a call at the end of your application that reboots your machine by calling the BIOS boot code and, hence, reinitializes all of memory.

- There is some overhead involved when invoking subroutines that remain resident through INT 27. Along with the resident portion of your code, you will have a copy of the DOS environment, which is a minimum of 128 bytes, and a PSP (program segment prefix), which is 256 bytes.

Inside the BASIC Data Segment

For relatively short assembly language programs, loading your subroutines into an integer array within BASIC is the easiest and safest way to store them.

Features

- Because you are working within BASIC's data segment, it is not necessary to address an external segment for your subroutine with the DEF SEG statement.

- By loading the subroutine inside BASIC's data segment and into a data area allocated and maintained by BASIC, there is no danger of your program stepping onto critical space inside or outside BASIC, or code outside BASIC stepping on your data.

Considerations

- If your BASIC application uses most of the available data space within BASIC's data segment, this is not the best method since you will be using additional data space by assigning the subroutine to an integer array.

- In certain configurations, this method is your only option. You must calculate how much free memory you have in your machine before you choose your method. Factors you must consider are: the size of DOS, the size of BASIC, whether your application needs Disk BASIC or Advanced BASIC, and the size of any resident routines such as MODE, PRINT, or GRAPHICS.

  The sizes of DOS and BASIC vary depending on the level of code you are using. On a small system, when you invoke BASIC, there may not be any free space in the top of memory. In this case, use the integer array method.

- The assembler code must be self-relocatable; that is, it must not contain any references to a paragraph number of a segment whose value depends on where

the subroutine was loaded. This means that all offsets must be expressed in relative terms so that your subroutine is able to resolve all references within itself.

Beyond the BASIC Data Segment

You must have at least 128K in your system to use this procedure. Also, you may be required to do some of your own memory management to ensure the integrity of your program. For these reasons, loading subroutines beyond the BASIC data segment is the least desirable method for assembly language interface.

Features

- You do not have to use space within BASIC's data segment for your routines.
- With systems that have a large amount of memory, you can load subroutines that are too large to fit within the BASIC data segment.

Considerations

When you load a subroutine outside BASIC's data segment, you are responsible for managing the area of memory occupied by your subroutine. To ensure the integrity of your program:

- You can restrict the size of BASIC's total addressable space with the CLEAR statement or the **/M:** switch from the BASIC command line. This ensures that none of BASIC's data overlays your subroutine.
- You must calculate within your program the new segment ID, which you will pass to the DEF SEG statement.

- When calculating the amount of available memory, you must allow for any routines that are resident at the time your subroutine is called (PRINT, MODE, or a virtual disk program, for example).

How to Load and Call Your Assembly Language Subroutines

The following are offered as examples of how assembly language subroutines are loaded into memory. There are essentially three ways:

- Poking or assigning it into memory from your BASIC program.
- BLOADing it from a file on disk or cassette.
- Loading it from a file on disk as a resident part of DOS.

Poking or Assigning a Subroutine into Memory

You can POKE or assign assembly language subroutines directly into memory from your program. In this way the subroutine actually becomes a part of your BASIC program.

Features

- You do not need an assembler to use this method.
- All code is contained in one file. You need not be concerned with the creation and maintenance of two or more separate files.

Considerations

- Coding each statement of your subroutine in hexadecimal can become a very tedious and error-prone procedure. This method is most useful for very short subroutines.

- If you poke your subroutine into a memory area outside of BASIC's data segment, you must perform any memory management necessary to reserve or protect subroutine space from BASIC or DOS, in order to ensure the integrity of your program.

Assigning a Subroutine into Memory

This is the procedure you use to assign machine code to an integer array and invoke the subroutine:

1. Determine the machine code for your subroutine. See "DEBUG" in *Disk Operating System* (DOS) manual under the "Assemble" option.

2. Dimension an integer array to the size you need. Remember, an integer uses 2 bytes of memory, so the size of your array is one-half the number of bytes in your subroutine.

   There are several reasons for using an integer array (2 bytes per element) instead of a single-precision (4 bytes per element) or a double-precision array (8 bytes per element). When you assign the subroutine to an integer array, you will never leave more than 1 byte of memory unused; with a double-precision array, you could waste 7 bytes. A fundamental reason for the integer array method is that you perform WORD assignment to the array elements rather than byte assignment, and a WORD happens to be 2 bytes. It would be easy to make a mistake if you were assigning four words to each array element since the CPU in the system expects to see the machine code in a low-byte, high-byte format. The

example that follows shows how the machine code in the assignment statements is switched from the order of the bytes as they appear in the assembled listing.

3. Define all scalars to be used in the application. After an array is dimensioned, any scalar created pushes or relocates the array farther up in memory. See the memory map in Appendix D for related information.

4. Assign the machine language code to the array elements, one word at a time. Remember that your system reads bytes expecting the low byte first, so store your data low-byte, high-byte.

5. Determine the offset of the subroutine within BASIC's data segment by using the VARPTR function.

6. Invoke the subroutine with the CALL statement or the USR function.

Example:

```
10 DEFINT A-Z: OPTION BASE 1: DIM ARRAY (3)
20 DATA &HCD55            :REM 55H Push BP
30 DATA &H5D05            :REM CD05H INT 5
40                        :REM 5DH POP BP
50 DATA &H90CB            :REM 90H NOP
70 FOR I = 1 TO 3: READ ARRAY(I): NEXT I
80 PRINT "This is an example of how "
90 PRINT "a BASIC routine can cause "
100 PRINT "a hard copy to be dumped "
110 PRINT "to a printer by using the "
120 PRINT "shift-print screen function "
130 PRINT "called via an assembly language "
140 PRINT "subroutine that is built into "
150 PRINT "a BASIC integer array"
160 SUBRT = VARPTR(ARRAY(1)): CALL SUBRT
```

Notice that the data statement reads data into the array a WORD at a time. The assembled listing would list the bytes in the order that they appear in the commented section of the program, yet the program switches the order of the bytes in the words as they are assigned to the array elements since the format of a word is low-byte, high-byte.

It is a recommended practice to always put VARPTR and CALL in the same line so CALL is never performed without locating the array containing the subroutine.

Poking a Subroutine into Memory

This next example uses the same assembly language subroutine, but the BASIC POKE statement is used to POKE the bytes into the array. The POKE statement takes care of the byte order for you, so you do not have to the order of the bytes as they would appear in an assembled listing.

```
10 DEFINT A-Z:OPTION BASE 1:P=0:I=0:J=0:
   DIM ARRAY(3)

20 DATA &H55            :REM 55H Push BP
30 DATA &HCD, &H05      :REM CD05H INT 5
40 DATA &H5D            :REM 5DH POP BP
50 DATA &HCB            :REM CBH RET FAR
60                      :REM 90H NOP
70 P=VARPTR(ARRAY(1)):FOR I=0 TO 4:READ J:
   POKE(P+I),J:NEXT I
80 PRINT "This is an example of how "
90 PRINT "a BASIC routine can cause "
100 PRINT "a hard copy to be dumped "
110 PRINT "to a printer by using the "
120 PRINT "shift-print screen function "
130 PRINT "called via an assembly language "
140 PRINT "subroutine that is built into "
150 PRINT "a BASIC integer array"
160 SUBRT = VARPTR(ARRAY(1)): CALL SUBRT
```

POKE can be used to put data anywhere in memory. To POKE this routine into another segment, line 70 would be:

```
DEF SEG = SEGMENT: FOR I=0 TO 4: READ J:
POKE(I,J): NEXT
```

Because the subroutine is placed in an area other than BASIC's data segment, there is no VARPTR reference to obtain the offset of the subroutine.

BLOADing the Subroutine from a File

You use the BASIC BLOAD command to load a memory image file directly into memory. The file can be a memory image file that was saved using the BASIC BSAVE command; or it can be an assembly language subroutine that was assembled with a BLOAD header, linked and converted to a .COM file using EXE2BIN. See *Disk Operating System* (the

DOS manual) for information on EXE2BIN. Assembly language subroutines can be BLOADed into an integer array or beyond BASIC's data segment.

Features

- When a subroutine is loaded into BASIC's data segment, BASIC performs all memory management. You are not responsible for the integrity of your subroutine once it is under the control of BASIC.

- By loading subroutines outside of BASIC's data segment it is possible to construct a program larger than BASIC's normal 64K limit.

Considerations

- An assembler is necessary in order to use this method.

- For an assembly language subroutine to be BLOADed into memory, it must appear to have been created by BSAVE. This is done by including a 7-byte BSAVE header that is linked as the first segment of the subroutine. These 7 bytes function as a loader that BLOAD looks at and then discards. The beginning of your routine in memory is the first byte following this header.

The example that follows shows how a subroutine is BLOADed into an integer array within BASIC and then is called from BASIC. The subroutine adds the contents of the first and second parameters and puts the results into the third parameter.

This also illustrates self-relocating code. A step-by-step explanation of the procedure follows.

1. Assemble and link your subroutine.

2. Convert the file to a .COM file via EXE2BIN.

3. From BASIC, dimension an integer array to the size needed. (Remember, size of the array subscript is 1/2 the number of bytes in your subroutine.)

4. Declare all scalars used in your application. This includes any parameters you may pass on through the CALL statement.

5. Obtain the offset of the array within BASIC's data space using the VARPTR function.

6. BLOAD the subroutine into the array.

7. Call the subroutine with this offset.

A Sample Subroutine

The function of this subroutine is to add the contents of the first and second parameters and put the results into the third parameter. It shows a unique way to pass parameters to a subroutine that is located inside an integer array. Parameters are passed by storing them as elements of an array into which the subroutine has been BLOADed.

The comments inserted in the program listing are intended to help you understand the most important points of the operation and structure of this subroutine. A more detailed analysis follows the listing.

```
AAAA                SEGMENT PARA PUBLIC 'CODE'
          DB        ØFDH            ;BSAVE id
          DW        Ø,Ø
          DW        TRAILER-HEADER ;Module size
AAAA      ENDS
          PAGE
          BASDATSEG SEGMENT BYTE PUBLIC 'CODE'
          ASSUME CS:BASDATSEG,DS:BASDATSEG
HEADER EQU    $    ;Start of memory image
SUBRT  PROC   FAR
       NOP         ;Adjusts for proper
;                   boundary also allows
;                   replacement by ØCCH,
;                   the INT 3, which is
;                   DEBUG's breakpoint.
       CALL   SKI  ;This pushes offset
;                   of  'BASE' into stack
BASE:              ;local workarea
WORKA  DW Ø ;First input parameter,
;                   BASIC's ARRAY(3)
WORKB  DW Ø ;Second input parameter,
;                   BASIC's ARRAY(4)
WORKC  DW Ø ;Result, BASIC's ARRAY(5)
SKIP:  POP    BX   ;Now BX knows at what
;                   offset into BASIC's
;                   data space is 'BASE'
       MOV    AX,WORKA - BASE[BX] ;Get 1st parm
       ADD    AX,WORKB - BASE[BX] ;Add 2nd parm
;                                   to ACC
       MOV    WORKC-BASE[BX],AX ;Pass to 3rd parm
       RET
SUBRT     ENDP
TRAILER   EQU      $       ;End of memory image
BASDATSEG ENDS
          END
```

Sample Subroutine Explanation

This example generates a header that looks as if it were created by BSAVE. It must be linked to the beginning of the load module by using a segment name that comes first alphabetically.

```
AAAA                SEGMENT PARA PUBLIC 'CODE'
          DB        ØFDH            ;BSAVE id
          DW        Ø,Ø
          DW        TRAILER-HEADER ;Module size
AAAA      ENDS
          PAGE
```

This value for module size, divided by 2, is the value to be substituted for N in the BASIC program below.

The BYTE alignment of the following SEGMENT statement is needed so the memory image will be loaded as the next byte following the dummy BSAVE header with no alignment padding bytes in between.

```
BASDATSEG SEGMENT BYTE PUBLIC 'CODE'
```

This code is to be loaded into a BASIC integer array in BASIC's data segment. The offsets shown by the assembler listing are relative to the start of the subroutine, not to the start of BASIC's data segment, which is what CS and DS really point to. The subroutine must do its own offset fixup by finding where in BASIC's data segment it is loaded, and then modifying the assembler's offsets by this base amount.

```
          ASSUME CS:BASDATSEG,DS:BASDATSEG
HEADER EQU     $     ;Start of memory image
SUBRT  PROC    FAR
       NOP           ;Adjusts for proper
;                     boundary also allows
;                     replacement by ØCCH,
;                     the INT 3, which is
;                     DEBUG's breakpoint.
       CALL    SKI   ;This pushes offset
;                     of  'BASE' into stack
BASE:                ;local workarea
WORKA  DW Ø ;First input parameter,
;                     BASIC's ARRAY(3)
WORKB  DW Ø ;Second input parameter,
;                     BASIC's ARRAY(4)
WORKC  DW Ø ;Result, BASIC's ARRAY(5)
SKIP:  POP     BX    ;Now BX knows at what
;                     offset into BASIC's
;                     data space is 'BASE'
       MOV     AX,WORKA - BASE[BX] ;Get 1st parm
       ADD     AX,WORKB - BASE[BX] ;Add 2nd parm
;                                    to ACC
```

Note how local storage is to be addressed. Adding the BX performs the needed self-relocation, changing local offsets to offsets from DS (BASIC's data segment).

```
       MOV     WORKC-BASE[BX],AX ;Pass to 3rd parm
       RET
SUBRT     ENDP
TRAILER   EQU      $       ;End of memory image
BASDATSEG ENDS
          END
```

The calling routine from BASIC would be as follows:

```
10 OPTION BASE 1
20 DEFINT A-Z
30'Define scalars before DIM
40 SUBRT = 0
50 DIM ARRAY(10)
60'Load subroutine into array
70 BLOAD "SUB2.BLO",SUBRT
80'Pass parameters into array
90 ARRAY(3)=2: ARRAY(4)=3
100'Find where array starts
110 SUBRT=VARPTR(ARRAY(1))
120'Parms are in the array
130 CALL SUBRT
140'Display result
150 PRINT ARRAY(5)
```

The next example performs the same function as the previous example. However, parameters are passed by putting variables names in the parenthesized list in the CALL statement.

```
10 OPTION BASE 1
20'Define scalars before DIM
30 SUBRT%=0: PARMC%=0
40 DIM ARRAY%(12)
50'Find start of array
60 SUBRT%=VARPTR(ARRAY%(1))
70'Load routine into array
80 BLOAD "SUB1.BL0", SUBRT%
90'Pass parms to subroutine
100 PARMA% = 2: PARMB% = 3
110'Locate subroutine
120 SUBRT% = VARPTR(ARRAY%(1))
130'Parms in CALL
140 CALL SUBRT%(PARMA%, PARMB%, PARMC%)
150'Display results
160 PRINT PARMC%
```

The subroutine named SUB1.BLO is as follows:

```
PARM    STRUC        ;Description of parameter list
SAVEBP  DW       Ø   ;Saves BASIC's BP register
RETOFF  DW       Ø   ;Offset of where to RET to
RETSEG  DW       Ø   ;Segment to return to
PARMC   DW       Ø   ;Offset to third parameter
PARMB   DW       Ø   ;Offset to second parameter
PARMA   DW       Ø   ;Offset to first parameter
PARM    ENDS

AAA     SEGMENT  PARA
        DB       ØFDH            ;BSAVE ID
        DW       Ø,Ø
        DW       TRAILER-HEADER  ;Module size
AAA     ENDS

BASDATSEG  SEGMENT  BYTE  PUBLIC 'CODE'
        ASSUME  CS:BASDATSEG,DS:BASDATSEG

HEADER  EQU   $    ;Start of memory image
SUBRT   PROC  FAR
        NOP        ;For debugging, can be
;                   replaced by  ØCCH (INT 3)
;                   which is the DEBUG
;                   breakpoint
        PUSH   BP   ;Save BASIC's BP register
        MOV    BP,SP  ;Set addressability to
;                       parm area on stack
        MOV    SI,[BP].PARMA  ;Get offset to parm1
        MOV    AX,[SI]  ;Get value of parm1
        MOV    SI,[BP].PARMB  ;Get offset to parm2
        MOV    AX,[SI]  ;Add value of parm2 to acc
        MOV    DI,[BP].PARMC  ;Get ofset to parm3
        MOV    [DI],AX  ;Pass result to parm3
        POP    BP   ;Restore BASIC's BP
        RET    6    ;Return to BASIC,
;                   throwing away the 3 parms
SUBRT   ENDP
TRAILER EQU    $    ;End of memory image
BASDATASEG ENDS
        END
```

Load your assembly language program into this external data area using the BASIC BLOAD command. You can also POKE the subroutine into memory. Load your subroutine beginning at offset Ø

of the first paragraph after the BASIC data segment. This segment ID can be calculated by adding to the value of BASIC's segment ID the maximum number of paragraphs that BASIC can address.

BASIC is not necessarily located in the same segment on every machine, so calculate where to BLOAD your subroutine using the following procedure:

- Set the current segment at segment Ø with the DEF SEG statement. The segment ID of BASIC's data segment is located in low memory, segment Ø, offsets &H51Ø and &H511. This memory area is mapped in the IBM Personal Computer *Technical Reference* manual in Section 3 under "Low Memory Maps."

- PEEK at these two values and assign the result to a variable.

- Add to this value the maximum number of paragraphs that BASIC will address. By default, this is 4Ø96 or &H1ØØØ. This value can be modified by the CLEAR statement or the **/M:** switch.

- Declare this value as your new segment ID with the DEF SEG statement.

- BLOAD your subroutine at offset Ø into this segment.

The BASIC code for this is:

```
1Ø DEF SEG = Ø  'Look at low memory
2Ø 'Get segment ID
3Ø V = PEEK(&H51Ø) + (256 * PEEK(&H511))
4Ø SEGID = V + &H1ØØØ'Add 4Ø96 paragraphs
5Ø 'Def seg to next segment after basic
6Ø DEF SEG = SEGID
7Ø 'Load into this segment at offset Ø
8Ø BLOAD "SUBROUT.COM" , Ø
```

Loading the Subroutine as a Resident Extension of DOS

You can load your assembled subroutine under DOS as though it were a command. BASIC can then call your subroutine when it is needed.

Features

- The area containing your subroutine is protected by DOS. You are not required to perform any memory management to ensure the integrity of your program.
- Your assembled code need not be self-relocatable.
- You can load very large modules with this method. DOS allocates the portion of memory necessary to load itself and its extensions before it allocates space to BASIC.
- Because your subroutine remains resident when loaded by DOS, you can write one subroutine that can be used by various programs. The subroutine need be loaded only once.

Considerations

- An assembler is necessary in order to use this method.
- DOS-loaded subroutines remain resident until the system is rebooted. If memory space is critical, you must code your application so that a reboot is performed when the application completes.

The DOS-loaded module (your assembled subroutine) consists essentially of two parts: the BASIC subroutine and the loader portion. Invoke the module as a DOS external command (type the filename with no extension). The following steps must happen in the load portion of your assembly language subroutine:

- Store in low memory the double-word vector pointing to the subroutine.
- Tell DOS how much of the module is to remain resident (just the BASIC subroutine portion, not the loader portion).
- Return to DOS through INT 27H, which leaves the subroutine resident.

A problem with the DOS-loaded module is that you must let BASIC know where the subroutine is now that it has been loaded. A vector in the first 1 K of memory that is reserved for users can be selected. The intra-application communication area (16 bytes at &H4FØ - 4FF in segment Ø) can also be used to contain the 4-byte vector that points to the subroutine. See the IBM Personal Computer *Technical Reference* in Section 3 under the "Low Memory Maps" tables.

The loader should store both the offset and the segment ID of the subroutine in a fixed location in segment Ø. BASIC can then execute a DEF SEG = Ø and a PEEK at the 4 bytes following offset &H4FØ to find where the subroutine is. This is similar to finding the value of BASIC's default DEF SEG mentioned earlier.

To avoid multiple copies of the subroutine becoming resident, define a .BAT file that invokes the subroutine to load it and invokes BASIC and its application. Upon completion, the application should call the BIOS boot code to reboot the system and therefore avoid leaving the subroutine in memory.

The .BAT file should appear as follows:

```
subroutine-name
BASIC application-name
```

The loader portion of the subroutine sets up the intra-application communication area pointer and leaves the subroutine resident:

```
SEGZERO SEGMENT AT Ø
        ORG 4FØH   ;Intra-application
;                       communication area
COMAREA_OFF  DW  ?     ;Vector pointing to
COMAREA_SEG  DW  ?     ;permanent resident
;                       BASIC subroutine.
SEGZERO ENDS

GRP     GROUP    AAAA,ZLOADER

AAAA    SEGMENT 'CODE' ;Identifies the start
;                           of load module
AAAA    ENDS           ;Forces class "code"
;                       to load before "data".

SUBDAT  SEGMENT PARA PUBLIC 'DATA'
;Insert local data to be used by subroutine here
SUBDAT  ENDS
```

```
SUBSEG  SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:SUBSEG
SUBRT   PROC    FAR     ;BASIC entry point
        PUSH    BP      ;Save BASIC's registers
        MOV     BP,SP   ;Get
;                        addressability
;                        to parms on stack
        PUSH    ES      ;Save BASIC'S
;                        registers
        MOV     AX,SUBDAT
        MOV     ES,AX   ;Set
;                        addressability
;                        to local data
        ASSUME  ES:SUBDAT ;DS still points
;                        to BASIC's data area
;Insert the BASIC subroutine here
        POP     ES      ;Restore BASIC's
        POP     BP      ;Registers
        RET     n       ;Return to BASIC,
;                        discard parms
;                        from stack
SUBRT   ENDP
SUBSEG  ENDS

ZLOADER SEGMENT BYTE 'ZLOAD'
        ASSUME  CS:ZLOADER,SS:ZSTACK
;                        Stack not available
;                        to subroutine
ENDRES  EQU     $           ;End BASIC subroutine.
;                            The rest
;                            of this subroutine
;                            does not stay resident
LOADER  PROC    FAR         ;Entry from DOS
        PUSH    ES          ;Set up stack to
;                            contain vector so
        XOR     AX,AX       ;'RET' will return
;                            to INT 27H
;                            instruction at
;                            offset Ø in PSP.
```

```
        PUSH    AX
        MOV     BYTE PTR ES:1,27H
;                       Change INT 20H
;                       to INT 27H at start
;                       of PSP
        XOR     AX,AX ;Clear a register
        MOV     ES,AX ;Set extra segment
                       to 0
        ASSUME  ES:SEGZERO
;                      ;Seg reg points to
;                       low memory
        MOV     COMAREA_OFF, OFFSET SUBRT
;                       Pass offset to entry
;                       point of subroutine
        MOV     COMAREA-SEG,SEG SUBRT
;                       Pass subroutine
;                       segment
        MOV     DX,OFFSET GRP:ENDRES+100H
;                       Pass offset of end
;                       of resident code.
;                       At exit, CS will
;                       point to the PSP
;                       which has the exit
;                       instruction 27H.
;                       This sets DX to the
;                       offset from the CS
;                       that is the end of
;                       the resident code.
        RET            ;Return to DOS, via
;                       INT 27H in PSP
;                       leaving subroutine
;                       resident.
LOADER  ENDP
ZLOADER ENDS
ZSTACK  SEGMENT PARA STACK 'STACK'
        DB      16 DUP ("STACK")
                      ;Stack used during
;                       procedure only
ZSTACK  ENDS
        END     LOADER
```

The subroutine with its loader is assembled and linked. It is then ready for execution as an .EXE external DOS command.

Once the BASIC application is loaded, it can find the subroutine by:

```
20 DEFINT A-Z
30 DEF SEG=0 'Look at low memory at
40 'inter application communication area.
50 SUBOFF=PEEK(&H4F0)+(256*PEEK(&H4F1))
60 SUBSEG=PEEK(&H4F2)+(256*PEEK(&H4F3))
```

It can then call the subroutine by:

```
100 DEF SEG = SUBSEG
110 CALL SUBOFF(PARMA,PARMB,PARMC)
120 DEF SEG
```

When finished with execution, the application can reboot the system, thus removing the subroutine, by:

```
1000 DEF SEG = &HF000: SUBOFF =      &HFFF0:
     CALL SUBOFF
```

How BASIC Interfaces with Assembly Language Subroutines

BASIC provides two interfaces for calling assembly language subroutines from your application. They are the CALL statement and the USR function. The CALL statement is the recommended procedure and all examples use this interface. See "USR Function" in the *BASIC Reference*.

When you invoke your subroutine the following is true:

- At entry, the segment registers DS, ES, and SS are set to the same segment value, the segment ID of BASIC's data segment. This is the default value for DEF SEG.

- At entry, the CS register contains the segment ID of the latest value specified for DEF SEG. By default, this is BASIC 's segment ID.

- At entry, the stack pointer register (SP) indicates that you have a minimum of eight words available on the stack for use by your subroutine. Other

space is provided on the stack for use by interrupts (such as TIMER) and DOS or BIOS calls invoked by the assembler subroutine. If your subroutine needs more stack space, create your own stack with a minimum of 128 bytes reserved for system usage above the requirements of the subroutine itself. Upon returning to BASIC, the stack segment (SS) and the stack pointer (SP) registers must have the same values as when the subroutine was called from BASIC. Care must be taken to preserve these values if you set up your own stack.

If the input parameter is a string, the value passed is the offset in BASIC's data segment of a 3-byte area called the string descriptor.

Byte Ø contains the length of the string (Ø to 255)

Byte 1 contains the lower 8 bits of the offset of the string in the string space area of BASIC's data segment.

Byte 2 contains the higher 8 bits of the offset of the string in the string space area of BASIC's data segment.

Warning: The subroutine must not change the contents of any of the 3 bytes of the *string descriptor*.

The subroutine can change the *content* of the string itself, but not its *length*.

If the subroutine changes a string, this may modify your program. The following example may change the string "TEXT" in the BASIC program.

```
A$ = "TEXT"
CALL SUBRT(A$)
```

However, the next example does not change the program, because the string concatenation causes

BASIC to copy the string into the string space where it can be safely changed without affecting the original text.

```
A$ = "BASIC" + ""
CALL SUBRT(A$)
```

To return from your subroutine, you must:

- Enable any interrupts disabled by the subroutine.
- Use an intersegment RET instruction since any subroutine call from BASIC is a FAR call.
- Restore all segment registers and the stack pointer (SP). All other registers and flags can be altered.

The CALL Statement

Execution of a CALL statement causes the following to occur:

- For each parameter in the variable list, the variable's location is pushed onto the stack. The location is specified as a 2-byte offset into BASIC's data segment. You can return values to BASIC through the parameters by changing the values of the variables pointed to by the parameter list.

  The CALLed subroutine must know how many parameters were passed. Parameters are referenced by adding a positive offset to BP after the called subroutine moves the current stack pointer into BP. The first instruction in your subroutine should be:

```
PUSH BP              ;SAVE BP
MOV BP, SP           ;MOVE SP TO BP
```

The offset into the stack of any one parameter is calculated as follows:

```
OFFSET from BP = 2 *(N - M) + 6
```

where N is the total number of parameters passed and M is the position of the specific parameter in the parameter list on the CALL statement. M can range from 1 to N.

In the Macro Assembler, the "STRUC" pseudo-op is a conventional way of defining the contents of the stack, showing the location of parameters.

```
FRAME     STRUC
SAVE BP   DW      ?    ;Saved BASIC's bp
RET OFF   DW      ?    ;Offset to RET to BASIC
RET SEG   DW      ?    ;Seg to RET to BASIC
PARMN     DW      ?    ;Offset to Nth parameter
          .
          .
          .
PARM1     DW      ?    ;Offset to first parameter
FRAME     ENDS

PARMSIZE EQU OFFSET PARM1-OFFSET RETSEG ;
  (To be used as operand on RET)
```

Warning: You must make sure that the parameters in the CALL statement match in number, type, and length the parameters expected by the subroutine; for instance, double-precision (8-byte) values.

- Control is transferred to the assembly language subroutine using the segment specified in the first DEF SEG statement and the offset specified in the CALL statement.

- The return segment ID specified in the CS register and the offset are pushed onto the stack.

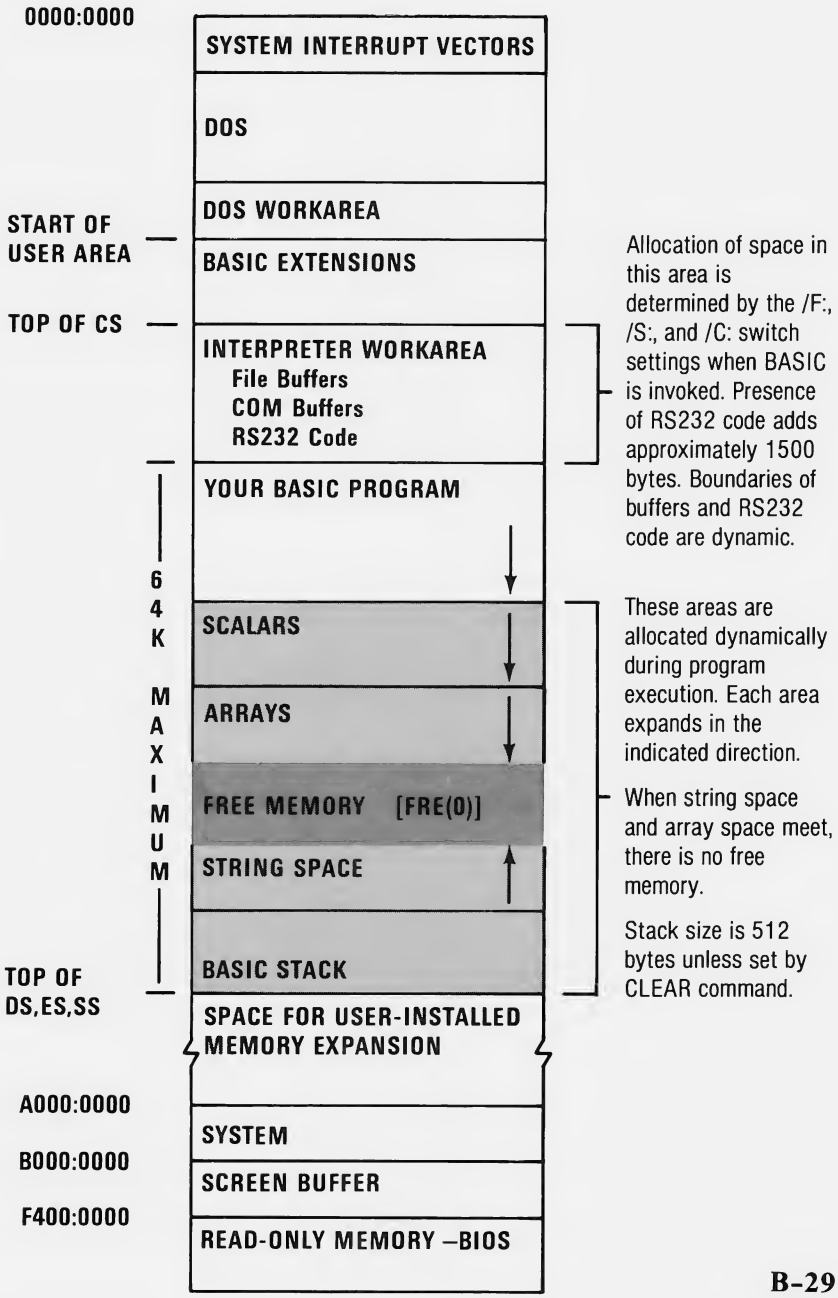
- When you call a subroutine using a CALL statement that specifies parameters, the subroutine must return with a RET N, where N is twice the

number of parameters in the list. This adjusts the stack to the condition at the start of the calling sequence.

- The CALL statement does not always have parameters. Also, if your routine is located in an integer array, it is possible to pass parameters in the array itself instead of through the CALL statement.

Memory Map

This is a memory map for Disk and Advanced BASIC. DOS and the BASIC extensions are not present for Cassette BASIC. Addresses are in the hexadecimal form SEGMENT:OFFSET.



Appendix C. Communications

This appendix describes the BASIC statements required to support RS232 asynchronous communication with other computers and peripherals.

Opening a Communications File

OPEN "COM... allocates a buffer for input and output in the same fashion as OPEN for disk files. See "OPEN "COM....

Communication I/O

Since each communications adapter is opened as a file, all input/output statements that are valid for disk files are also valid for communications.

Communications sequential input statements are the same as those for disk files. They are:

INPUT #
LINE INPUT #
INPUT$

Communications sequential output statements are also the same as those for disk files, and are:

PRINT #
PRINT # USING
WRITE #

See "INPUT" and "PRINT" for details of coding syntax and usage.

GET and PUT for Communications Files

GET and PUT are only slightly different for communications files than for disk files. They are used for fixed length I/O from or to the communications file. Instead of specifying the record number to be read or written, you specify the number of bytes to be transferred into or out of the file buffer. This number cannot exceed the value set by the LEN option on the OPEN "COM... statement. See "GET" and "PUT."

I/O Functions

The most difficult aspect of asynchronous communication is processing characters as fast as they are received. At rates of 12ØØ bps or higher, it may be necessary to suspend character transmission from the other computer long enough to "catch up." This can be done by sending XOFF (CHR$(19)) to the other computer and XON (CHR$(17)) when you are ready to resume. XOFF tells the other computer to stop sending, and XON tells it to start sending again.

> **Note:** This is a commonly used convention, but it is not universal. Whether it is valid depends on the protocol implemented between you and the other computer or peripheral.

Disk BASIC and Advanced BASIC provide three functions that help determine when an "overrun" condition is likely to occur. These are:

LOC(f)
LOF(f)
EOF(f)

> **Note:** A **Communication buffer overflow** can occur if a read is attempted after the input buffer is full (that is, when LOF(f) returns Ø).

INPUT$ Function

The INPUT$ function is preferred over the INPUT # and LINE INPUT # statements when reading communications files, since all ASCII characters may be significant in communications. INPUT # is least desirable because input stops when a comma or carriage return is seen. LINE INPUT # stops when a carriage return is seen.

INPUT$ allows all characters read to be assigned to a string. INPUT$(*n,f*) returns *n* characters from the #*f* file. The following statements are efficient for reading a communications file:

```
1Ø WHILE NOT EOF(1)
2Ø A$=INPUT$(LOC(1),#1)
   .
   .
   .
(process data returned in A$)
   .
   .
   .
1ØØ WEND
```

When there are characters in the buffer, line 2Ø assigns them to A$, and they are processed. If there are more than 255 characters in the buffer, only 255 are returned at a time to prevent **String overflow**. Since EOF(1) is false, input continues until the input buffer is empty. This procedure is simple, concise, and fast.

To process characters quickly, avoid examining every character as you receive it. If you are looking for special characters (such as control characters), you can use the INSTR function to find them in the input string.

A Sample Program

The following program allows the IBM Personal Computer to be used as a conventional "dumb"

terminal in a full duplex mode. This program assumes a 3ØØ bps line and an input buffer of 256 bytes (the /C: option was not used in the BASIC command).

```
1Ø REM   dumb terminal example
2Ø 'set screen to monochrome text mode
3Ø '    and set width to 4Ø
4Ø SCREEN Ø,Ø: WIDTH 4Ø
5Ø 'turn off key display; clear screen
6Ø '   make sure all files are closed
7Ø KEY OFF: CLS: CLOSE
8Ø 'define numeric variables as integer
9Ø DEFINT A-Z
1ØØ 'define true and false
11Ø FALSE=Ø: TRUE=NOT FALSE
12Ø 'define the XON, XOFF characters
13Ø XOFF$=CHR$(19): XON$=CHR$(17)
14Ø 'open communications to file number 1,
15Ø ' 3ØØ bps, EVEN parity, 7 data bits
16Ø OPEN "COM1:3ØØ,E,7" AS #1
17Ø 'use screen as a file, just for fun
18Ø OPEN "SCRN:" FOR OUTPUT AS #2
19Ø 'turn cursor on
2ØØ LOCATE ,,1
4ØØ PAUSE=FALSE: ON ERROR GOTO 9ØØØ
49Ø '
5ØØ 'send keyboard input to com line
51Ø B$=INKEY$: IF B$<>"" THEN PRINT #1,B$;
52Ø 'if com buffer is empty, check key in
53Ø IF EOF(1) THEN 51Ø
54Ø 'if buffer more than 1/2 full, then
55Ø 'set PAUSE flag to say input suspended,
56Ø 'send XOFF to host to stop transmission
57Ø IF LOC(1)>128 THEN PAUSE=TRUE
575 PRINT #1,XOFF$
```

```
580 'read contents of com buffer
590 A$=INPUT$(LOC(1),#1)
600 'remove linefeeds to avoid double spaces
610 'when input displayed on screen
620 LFP=0
625 'look for linefeed
630 LFP=INSTR(LFP+1,A$,CHR$(10))
640 IF LFP>0 THEN MID$(A$,LFP,1)=" "
645 GOTO 630
650 'display com input, and check for more
660 PRINT #2,A$;: IF LOC(1)>0 THEN 570
670 'if transmission suspended by XOFF,
680 'resume by sending XON
690 IF PAUSE THEN PAUSE=FALSE
695 PRINT #1,XON$;
700 'check for keyboard input again
710 GOTO 510
8999 'if error, print its number and retry
9000 PRINT "ERROR NO.";ERR: RESUME
```

Notes on the Program

- "Asynchronous" communication implies character I/O as opposed to line or block I/O. Therefore, all PRINTs (either to communications file or to screen) are ended with a semicolon. This stops the carriage return normally issued at the end of the list of values to be printed.

- Line 90, where all numeric variables are defined as integers, is coded because any program looking for speed optimization should use integer counters in loops where possible.

- Note in line 510 that INKEY$ will return a null string if no character is pending.

Operation of Control Signals

The output from the Asynchronous Communications Adapter conforms to the Electronic Industries Association (EIA) RS232-C standard for interface between Data Terminal Equipment (DTE) and Data Communications Equipment (DCE). This standard defines several control signals that are transmitted or received by your IBM Personal Computer to control the interchange of data with another computer or peripheral. These signals are DC voltages that are either ON (greater than +3 volts) or OFF (less than -3 volts). See the IBM Personal Computer *Technical Reference*.

Control of Output Signals with OPEN

When you start BASIC on your IBM Personal Computer, the RTS (Request To Send) and DTR (Data Terminal Ready) lines are held off. When an OPEN "COM... statement is performed, both of these lines are normally turned on. However, you can specify the **RS** option in the OPEN "COM... statement to suppress the RTS signal. The lines remain on until the communications file is closed (by CLOSE, END, NEW, RESET, SYSTEM, or RUN without the **R** option). Even if the OPEN "COM... statement fails with an error (as described below), the DTR line (and RTS line, if applicable) is turned on and stays on. This allows you to retry the OPEN without having to execute a CLOSE.

Use of Input Control Signals

Normally, if either the CTS (Clear To Send) or DSR (Data Set Ready) lines are off, then an OPEN "COM... statement does not execute. After one second, BASIC returns a **Device timeout** error (error

code 24). The Carrier Detect (sometimes called Receive Line Signal Detect) can be either on or off; it has no effect on the operation of the program.

However, you can specify how you want these lines tested with the **RS, CS, DS**, and **CD** options on the OPEN "COM... statement. See "OPEN"COM... Statement."

If any of the signals being tested are turned off while the program is executing, I/O statements associated with the communications file do not work. For example, when you execute a PRINT # statement after the CTS or DSR line is turned off, a **Device fault** (code 25) or **Device timeout** (code 24) error occurs. The RTS and DTR remain on even if such an error occurs.

You can test for a line disconnect by using the INP function to read the bits in the MODEM Status Register on the Asynchronous Communications Adapter. See the following section, "Testing for Modem Control Signals" for details.

Testing for Modem Control Signals

Four input control signals are picked up by the Asynchronous Communications Adapter. These signals are CTS and DSR (described previously) Carrier Detect (sometimes called Received Line Signal Detect) (pin 8), and Ring Indicator (pin 22). You can specify how you want to test the CTS, DSR, and CD lines with the OPEN "COM... statement. Ring Indicator is not used at all by the communications function in BASIC.

If you need to test for any of these signals in a program, you can check the bits corresponding to these signals in the MODEM Status Register on the Asynchronous Communications Adapter. To read the 8 bits in this register, use the INP function; INP(&H3FE) to read the register on an unmodified communications adapter; and INP(&H2FE) to read it

on a modified communications adapter. See the "Asynchronous Communications Adapter" section of the IBM Personal Computer *Technical Reference* manual for a description of which bits in the Status Register correspond to which control signals. You can also use the Delta bits in this register to determine if transient signals have appeared on any of the control lines. Note that for a control signal to have meaning, the pin corresponding to that signal must be connected in the cable to your modem or to the other computer.

You can also test for bits in the Line Status Register on the Asynchronous Communications Adapter. Use INP(&H3FD) to access this register on an unmodified communications adapter, and INP(&H2FD) to access it on a modified communications adapter. Again, the bits are described in the IBM Personal Computer *Technical Reference*. These bits can be used to determine what types of errors have occurred on receipt of characters from the communications line or whether a break signal has been detected.

Direct Control of Output Control Signals

You can control the RTS or DTR control signals directly from a BASIC program with an OUT statement. The on/off states of these signals are controlled by bits in the MODEM Control Register on the Asynchronous Communications Adapter. The address of this register is &H3FC on an unmodified communications adapter and &H2FC on a modified communications adapter. The IBM Personal Computer *Technical Reference* describes which of these bits correspond to which signals.

You can also change bits in the Line Control Register on the Asynchronous Communications Adapter. Be careful in modifying these bits because most of them have been set by BASIC when an OPEN statement is executed and changing a bit can cause communications failure. The Line Control Register is

at address &H3FB on an unmodified communications adapter and at address &H2FB on a modified communications adapter.

When changing bits in either the MODEM Control Register or the Line Control Register, first read the register (with an INP function), and then rewrite the register with only the pertinent bit or bits changed.

A bit you may wish to control in the Line Control Register is bit 6, the Set Break bit. This bit permits you to produce a Break signal on the communications send line. A Break is often used to signal a remote computer to stop transmission. Typically a Break lasts for 1/2 second. To produce such a signal, you must turn on the Set Break, wait for the desired time of the Break signal, and then turn the bit off. The following BASIC statements produce a Break signal of about 1/2 second duration on an unmodified communications adapter.

```
10 IC%=INP(&H3FB)
20 'get contents of modem register
30 IZ%=IC% OR &H4040 'turn ON the Set Break bit
50 OUT &H3FB,IZ%
60 'transmit to modem control register
70 FOR I=1 TO 500: NEXT I
80 'delay half a second
90 OUT &H3FB,IC% 'turn Set Break bit OFF in register
100 'turn Set Break bit OFF in register
```

Communication Errors

Errors occur on communication files in the following order:

1. When opening the file–

   a. **Device timeout** if one of the signals to be tested (CTS, DSR, or CD) is missing.

2. When reading data–

a. **Com buffer overflow** if overrun occurs.

b. **Device I/O error** for overrun, break, parity, or framing errors.

c. **Device fault** if you lose DSR or CD.

3. When writing data—

a. **Device fault** if you lose CTS, DSR, or CD on a Modem Status Interrupt while BASIC was doing something else.

b. **Device timeout** if you lose CTS, DSR, or CD while waiting to put data in the output buffer.

Appendix D. ASCII Character Codes

The following table lists all the ASCII codes (in decimal) and their associated characters. These characters can be displayed using PRINT CHR$(*n*), where *n* is the ASCII code. The column headed "Control Character" lists the standard interpretations of ASCII codes Ø to 31 (usually used for control functions or communications).

Each of these characters can be entered from the keyboard by pressing and holding the Alt key, then pressing the digits for the ASCII code on the numeric keypad. Note, however, that some of the codes have special meaning to the BASIC Program Editor. It uses its own interpretation for the codes and may not display the special character listed here.

ASCII Value	Character	Control Character	ASCII Value	Character
000	(null)	NUL	032	(space)
001	☺	SOH	033	!
002	☻	STX	034	,,
003	♥	ETX	035	#
004	♦	EOT	036	$
005	♣	ENQ	037	%
006	♠	ACK	038	&
007	(beep)	BEL	039	'
008	◘	BS	040	(
009	(tab)	HT	041	)
010	(line feed)	LF	042	*
011	(home)	VT	043	+
012	(form feed)	FF	044	,
013	(carriage return)	CR	045	-
014	♫	SO	046	.
015	☼	SI	047	/
016	►	DLE	048	0
017	◄	DC1	049	1
018	↕	DC2	050	2
019	!!	DC3	051	3
020	¶	DC4	052	4
021	§	NAK	053	5
022	▬	SYN	054	6
023	↨	ETB	055	7
024	↑	CAN	056	8
025	↓	EM	057	9
026	→	SUB	058	:
027	←	ESC	059	;
028	(cursor right)	FS	060	<
029	(cursor left)	GS	061	=
030	(cursor up)	RS	062	>
031	(cursor down)	US	063	?

ASCII Value	Character	ASCII Value	Character
064	@	096	`
065	A	097	a
066	B	098	b
067	C	099	c
068	D	100	d
069	E	101	e
070	F	102	f
071	G	103	g
072	H	104	h
073	I	105	i
074	J	106	j
075	K	107	k
076	L	108	l
077	M	109	m
078	N	110	n
079	O	111	o
080	P	112	p
081	Q	113	q
082	R	114	r
083	S	115	s
084	T	116	t
085	U	117	u
086	V	118	v
087	W	119	w
088	X	120	x
089	Y	121	y
090	Z	122	z
091	[	123	{
092	\	124	¦
093	]	125	}
094	^	126	~
095	_	127	⌂

ASCII Value	Character	ASCII Value	Character
128	Ç	160	á
129	ü	161	í
130	é	162	ó
131	â	163	ú
132	ä	164	ñ
133	à	165	Ñ
134	å	166	ª
135	ç	167	º
136	ê	168	¿
137	ë	169	⌐
138	è	170	¬
139	ï	171	½
140	î	172	¼
141	ì	173	¡
142	Ä	174	«
143	Å	175	»
144	É	176	░
145	æ	177	▒
146	Æ	178	▓
147	ô	179	│
148	ö	180	┤
149	ò	181	╡
150	û	182	╢
151	ù	183	╖
152	ÿ	184	╕
153	Ö	185	╣
154	Ü	186	║
155	¢	187	╗
156	£	188	╝
157	¥	189	╜
158	Pt	190	╛
159	ƒ	191	┐

ASCII Value	Character
192	└
193	┴
194	┬
195	├
196	─
197	┼
198	╞
199	╟
200	╚
201	╔
202	╩
203	╦
204	╠
205	═
206	╬
207	╧
208	╨
209	╤
210	╥
211	╙
212	╘
213	╒
214	╓
215	╫
216	╪
217	┘
218	┌
219	■
220	▬
221	▌
222	▐
223	▬

ASCII Value	Character
224	α
225	β
226	Γ
227	π
228	Σ
229	σ
230	μ
231	τ
232	Φ
233	Θ
234	Ω
235	δ
236	∞
237	∅
238	ϵ
239	∩
240	≡
241	±
242	≥
243	≤
244	⌠
245	⌡
246	÷
247	≈
248	°
249	•
250	·
251	√
252	ⁿ
253	²
254	■
255	(blank 'FF')

Extended Codes

For certain keys or key combinations that cannot be represented in standard ASCII code, an extended code is returned by the INKEY$ system variable. A null character (ASCII code 000) will be returned as the first character of a two-character string. If a two-character string is received by INKEY$, go back and examine the second character to determine the actual key pressed. Usually, this second code is the scan code of the primary key that was pressed. The ASCII codes (in decimal) for this second character and the associated key(s) are listed on the next page.

Second Code	Meaning
3	(null character) NUL
15	(shift tab) --<÷÷
16-25	Alt- Q, W, E, R, T, Y, U, I, O, P
30-38	Alt- A, S, D, F, G, H, J, K, L
44-50	Alt- Z, X, C, V, B, N, M
59-68	Function keys F1 through F10 (when disabled as soft keys)
71	Home
72	Cursor Up
73	Pg Up
75	Cursor Left
77	Cursor Right
79	End
80	Cursor Down
81	Pg Dn
82	Ins
83	Del
84-93	F11-F20 (Shift- F1 through F10)
94-103	F21-F30 (Ctrl- F1 through F10)
104-113	F31-F40 (Alt- F1 through F10)
114	Ctrl-Prtsc
115	Ctrl-Cursor Left (Previous Word)
116	Ctrl-Cursor Right (Next Word)
117	Ctrl-End
118	Ctrl-Pg Dn
119	Ctrl-Home
120-131	Alt- 1,2,3,4,5,6,7,8,9,0,-,=
132	Ctrl-Pg Up

Appendix E. Scan Codes

Key	Scan code in hex	Key	Scan code in hex
ESC	01	|← →|	0F
! 1	02	Q	10
@ 2	03	W	11
# 3	04	E	12
$ 4	05	R	13
% 5	06	T	14
^ 6	07	Y	15
& 7	08	U	16
* 8	09	I	17
(9	0A	O	18
) 0	0B	P	19
_ –	0C	{ [	1A
+ =	0D	}]	1B
←	0E	↵	1C

Key	Scan code in hex	Key	Scan code in hex
Ctrl	1D	¦ \	2B
A	1E	Z	2C
S	1F	X	2D
D	20	C	2E
F	21	V	2F
G	22	B	30
H	23	N	31
J	24	M	32
K	25	< ,	33
L	26	> .	34
: ;	27	? /	35
“ ’	28	⇧Right	36
~ ‘	29	PrtSc *	37
⇧Left	2A	Alt	38

Key	Scan code in hex	Key	Scan code in hex
Sp bar	39	7 Home	47
Caps Lock	3A	8 ↑	48
F1	3B	9 Pg Up	49
F2	3C	–	4A
F3	3D	4 ←	4B
F4	3E	5	4C
F5	3F	6 →	4D
F6	40	+	4E
F7	41	1 End	4F
F8	42	2 ↓	50
F9	43	3 PgDn	51
F10	44	0 Ins	52
Num Lock	45	. Del	53
Scroll Lock	46		

Glossary

absolute coordinates In computer graphics, a pair of values that specify the location of a point with respect to the origin of the coordinate system. Contrast with *relative coordinates.*

access mode A technique used to get a specific logical record from, or put a logical record into, a file.

accuracy The quality of being free from error. On a machine, this is actually measured and refers to the size of the error between the actual number and its value as stored in the machine.

active page On the Color/Graphics Monitor Adapter, the part of the screen buffer that has information written to it. It can be different from the part of the screen buffer whose information is being displayed.

adapter A mechanism for attaching parts.

address (noun) The location of a register, a particular part of memory, or some other data source or destination. (verb) To refer to a device or a data item by its address.

addressable point In computer graphics, any point in a display space that can be addressed. Such points are finite in number and form a discrete grid over the display space.

algorithm A set of well-defined rules for the solution of a problem in a finite number of steps.

allocate To assign a resource, such as a disk file or a part of memory, to a specific task.

alphabetic character A letter of the alphabet.

alphameric or alphanumeric Pertaining to a character set that contains letters and digits.

application program A computer program that accomplishes a specific task, such as word processing or processing payroll data.

argument A value that is passed from a calling program to a function.

arithmetic overflow Same as *overflow.*

array An arrangement of elements in a table format.

ASCII American National Standard Code for Information Interchange. The standard code used for exchanging information among data processing systems and associated equipment. The ASCII set consists of control characters and graphic characters.

asynchronous Without regular time relationship; unpredictable with respect to the execution of a program's instructions.

attribute A property or characteristic of one or more items.

background The area that surrounds the subject. In particular, the part of the display screen surrounding a character.

backup A system, device, file, or facility that can be used as an alternative in case of a malfunction or loss of data.

baud A unit of signaling speed equal to the number of discrete conditions or signal events per second.

binary Pertaining to a condition that has two possible values or states. Also, refers to the Base 2 numbering system.

bit A binary digit.

blank A part of a data medium in which no characters are recorded. Also, the space character.

blinking An intentional regular change in the intensity of a character on the screen.

boolean value A numeric value that is interpreted as "true" (if it is not zero) or "false" (if it is zero).

bootstrap An existing version, perhaps a primitive version, of a computer program that is used to establish another version of the program. Can be thought of as a program that loads itself.

bps Bits per second.

bubble sort A technique for sorting a list of items into sequence. Pairs of items are examined, and exchanged if they are out of sequence. This process is repeated until the list is sorted.

buffer An area of storage that is used to compensate for a difference in rate of flow of data, or time of occurrence of events, when transferring data from one device to another. Usually refers to an area reserved for I/O operations, into which data is read or from which data is written.

bug An error in a program.

byte The representation of a character in binary. Eight bits.

call To bring a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and jumping to an entry point.

carriage return character (CR) A character that causes the print or display position to move to the first position on the same line.

channel A path along which signals can be sent; for example, a data channel or an output channel.

character A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A connected sequence of characters is called a character *string*.

character device A device that is designed to do character I/O in a serial manner, like CON, AUX, and PRN.

clipping See *line clipping*.

clock A device that generates periodic signals used for synchronization. Each signal is called a clock pulse or clock tick.

code (verb) To represent data or a computer program in a symbolic form that can be accepted by a computer; to write a routine. (noun) Loosely, one or more computer programs, or part of a program.

comment An explanatory statement in a program. Comments include information that can be helpful in running the program or reviewing the output listing.

communication The transmission and reception of data.

complement An "opposite" number. In particular, a number that can be derived from a given number by subtracting it from another given number.

compression The moving of fragmented data into a contiguous region of memory, leaving other regions free for other data.

concatenation The operation that joins two strings together in the order specified, forming a single string with a length equal to the sum of the lengths of the two strings.

constant A fixed value or data item.

control character A character whose occurrence in a particular context starts, modifies, or stops a control operation. A control operation is an action that affects the recording, processing, transmission, or interpretation of data; for example, carriage return, font change, or end of transmission.

control data See *control character.*

coordinates Numbers that identify a location on the display.

current directory The default directory for each drive on a computer system. This is the directory that BASIC searches if you enter a filename without a path specification.

cursor A movable marker on the display screen that indicates where the next character will be entered, replaced, or deleted.

debug To find and eliminate mistakes in a program.

default A value or option that is assumed when none is specified.

delimiter A character that groups or separates words or values in a line of input, such as commas, colons, semicolons, and blanks.

device driver A program that interfaces input/output to a device.

diagnostic Pertaining to the detection and isolation of a malfunction or mistake.

directory A table of identifiers and references to the corresponding items of data. For example, the directory for a disk contains the names of files on the disk (identifiers), along with information that tells DOS where to find the file on the disk. See also *tree-structured directories.*

disabled A state that prevents the occurrence of certain types of interruptions.

DOS Disk Operating System. In this book, refers only to the IBM Personal Computer Disk Operating System.

double precision In the representation of numbers, the degree of accuracy that requires the use of two computer words. In double precision, numbers are stored with 17 digits of accuracy and printed with up to 16 digits. Contrast with *single precision.*

dummy Having the appearance of a specified thing but not having the capacity to function as such. For example, a dummy argument to a function.

duplex In data communication, pertaining to a simultaneous two-way independent transmission in both directions. Same as "full duplex."

dynamic Occurring at the time of execution.

echo To reflect received data to the sender. For example, keys pressed on the keyboard are usually echoed as characters displayed on the screen.

edit To enter, modify, or delete data.

element A member of a set; in particular, an item in an array.

enabled A state of the processing unit that allows certain types of interruptions.

end of file (EOF) A "marker" immediately following the last record of a file, signaling the end of that file.

environment A set of text strings, less than 32K bytes total, that conveys various configuration parameters.

event An occurrence or happening. In IBM Personal Computer Advanced BASIC, refers to the events tested by ON COM(n), ON KEY(n), ON PEN, ON PLAY(n), ON STRIG(n), and ON TIMER.

execute To perform a computer instruction or program.

expression A notation that has a value. Usually, a combination of variables, constants, and operators, such as X - 3.

fault An accidental condition that causes a device to fail to perform in a required manner.

field In a record, a specific area used for a particular category of data.

file A set of related records treated as a unit.

fixed disk The IBM nonremovable disk drive that has no drive cover or handle.

flag Any of various types of indicators used for identification; for example, a character that signals the occurrence of some condition.

font A family or assortment of characters of a particular size and style.

foreground The part of the display area that is the character itself.

format The particular arrangement or layout of data on a data medium, such as the screen or a disk.

form feed (FF) A character that causes the print or display position to move to the next page.

function A procedure that returns a value that depends on the value of one or more independent variables in a specified way.

function keys Keys on the computer keyboard that tell the system to perform certain commands. The keys F1-F10 on the keyboard.

graphic A symbol produced by a process such as handwriting, printing, or drawing.

half duplex In data communication, pertaining to an alternate, one way at a time, independent transmission.

hard copy A printed copy of machine output in a visually readable form.

hertz (Hz) A unit of frequency equal to one cycle per second.

hierarchy A structure having several levels, arranged in a tree-like form. "Hierarchy of operations" refers to the relative priority assigned to arithmetic or logical operations that must be performed.

host The primary or controlling computer in a multiple computer installation.

housecleaning An operation in which BASIC compresses string space by collecting all its useful data and frees up unused areas of memory that were once used for strings.

implicit declaration The establishment of a dimension for an array without it having been explicitly declared in a DIM statement.

increment A value used to alter a counter.

initialize To set counters, switches, addresses, or contents of memory to zero or other starting values at the beginning of, or at prescribed points in, the operation of a computer routine.

instruction In a programming language, any meaningful expression that specifies one operation and its operands, if any.

integer One of the numbers 0, ±1, ±2, ±3, ...

integrity Preservation of data for its intended purpose; data integrity exists as long as accidental or malicious destruction, alteration, or loss of data are prevented.

interface (noun) A shared boundary in which two systems interact.

interpreter A system program that is used to translate and execute each source language statement of a computer program before going on to the next statement.

interrupt To stop a process in such a way that it can be resumed.

intra-application communication area A 16-byte area in low memory starting at address H4F0 which is reserved for use by any application.

invoke To activate a procedure at one of its entry points.

joystick A computer graphics lever that can pivot in all directions and is used as a locator device.

justify To align characters horizontally or vertically to fit the positioning constraints of a required format.

K When referring to memory capacity, two to the tenth power (1024 in decimal notation).

keyword One of the predefined words of a programming language; a reserved word.

leading The first part of something; for example, leading zeros or leading blanks in a character string.

light pen A light-sensitive device that can be used instead of the keyboard to select a location on the screen.

line When referring to text on a screen or printer, one or more characters output before a return to the first print or display position. When referring to input, a string of characters accepted by the system as a single block of input; for example, all characters entered before you press the Enter key. In graphics, a series of points drawn on the screen to form a straight line. In data communications, any physical medium, such as a wire or microwave beam, that is used to transmit data.

line clipping A process in which points referenced outside a coordinate range become invisible in the viewing area. Any image crossing the viewing area (lying partially within and partially without) is cut off or "clipped" at the viewing area boundaries so that only points in range appear.

line feed (LF) A character that causes the print or display position to move to the corresponding position on the next line.

literal An explicit representation of a value, especially a string value; a constant.

location Any place in which data can be stored.

logical line A string of text between one Enter and another. It is treated by BASIC as a single unit.

loop A set of instructions that can be executed repeatedly while a certain condition is true.

M Mega; one million. When referring to memory, two to the twentieth power; 1,048,576 in decimal notation.

machine infinity The largest number that can be represented in a computer's internal format.

mantissa For a number expressed in floating point notation, the numeral that is not the exponent.

mapping The translation of coordinate values between the world coordinate system, as defined by the WINDOW statement, and the physical coordinate system of the viewport.

mask A pattern of characters that controls the retention or elimination of another pattern of characters.

matrix An array with two or more dimensions.

matrix printer A printer in which each character is represented by a pattern of dots.

megabyte 1,048,576 bytes. Same as two to the twentieth power.

menu A list of available operations. You select from the list the operation you want.

nest To incorporate a structure into another structure of the same kind. For example, you can nest loops within other loops or call subroutines from other subroutines.

notation A set of symbols, and the rules for their use, for the representation of data.

null Empty, having no meaning. In particular, a string with no characters in it.

octal Pertaining to a Base 8 number system.

offset The number of units from a starting point (in a record, control block, or memory) to some other point. For example, in BASIC the actual address of a memory location is given as an offset in bytes from the location defined by the DEF SEG statement.

on-condition An occurrence that could cause a program interruption. It can be the detection of an unexpected error, or of an occurrence that is expected, but at an unpredictable time.

operand An expression in an instruction that must be acted upon when the instruction is carried out.

operating system Software that controls the execution of programs; often used to refer to DOS.

operation A program step undertaken or executed by a computer.

operator A symbol that represents the action to be performed in a mathematical operation.

overflow The result of an operation that exceeds the capacity of the intended unit of storage.

overlay To use the same areas of memory for different parts of a computer program at different times.

overwrite To record into an area of storage so as to destroy the data that was previously stored there.

pad To fill a block with dummy data, usually zeros or blanks.

page Part of the screen buffer that can be displayed and/or written on independently.

palette In computer graphics, a range of colors.

parameter A variable that is given a constant value for a specified application. Or, a name in a procedure that refers to an argument passed to that procedure.

parity check A technique for testing transmitted data. Typically, a binary digit is appended to a group of binary digits to make the sum of all the digits either always even (even parity) or always odd (odd parity).

path A specified direction used to find a particular file. Used with directories and any command or statement that accepts a file specification.

peripheral device In a computer system, any equipment that provides the processing unit with outside communication.

physical coordinate system The logical limits of the screen. See "View" and "Window" statements in this manual.

pixel A point or location on a display screen that is used to form part of an image on the screen. Also, the bits that contain the information for that point.

port An access point for data entry or exit.

position In a string, each location that can be occupied by a character and that can be identified by a number.

precision A measure of the ability to distinguish between nearly equal values.

prompt A message or symbol that appears on the screen, asking for information from the user.

protect To restrict access to or use of all, or part of, a data processing system.

queue A line or list of items waiting for service; the first item that goes into the queue is the first item to be serviced.

random access memory Storage in which you can read and write to any desired location. Sometimes called "direct access storage."

range The set of values that a quantity or function can take.

raster scan A technique of generating a display image by a line-by-line sweep across the entire display screen. This is the way pictures are created on a television screen.

read-only A type of access that allows data to be read but not modified.

record A collection of related information treated as a unit. For example, in stock control, each invoice might be one record.

register A storage device with a specified capacity such as a bit, a byte, or a computer word.

relative coordinates In computer graphics, a pair of values that identify the location of a point by specifying displacements from some other point.

reserved word A word that is defined in BASIC for a special purpose and cannot be used as a variable name.

resolution In computer graphics, a measure of the sharpness of an image, expressed as the number of lines per unit length.

reverse image Highlighting a character field or cursor by reversing its color and its background.

root directory The directory that is created on each disk when it is formatted. Also called the "base" or "main" directory.

routine Part of a program, or a sequence of instructions called by a program, that can have some general or frequent use.

row A horizontal arrangement of characters or other expressions.

scalar A value or variable that is not an array.

scale To change the representation of a quantity, expressing it in other units, so that its range is brought within a specified range.

scaling In computer graphics, the process of WINDOW mapping world coordinates to physical coordinates. See "WINDOW Statement" in this manual.

scan To examine sequentially, part by part. See *raster scan.*

scroll To move all or part of the screen material up or down, left or right, to allow new information to appear.

segment A particular 64K-byte area of memory.

sequential access An access mode in which records are retrieved in the same order in which they were written. Each successive access to the file refers to the next record in the file.

single precision In the representation of numbers, the degree of accuracy that requires the use of one computer word. In single precision, seven digits are stored and up to seven digits are printed. Contrast with *double precision.*

stack A method of temporarily storing data so that the last item stored is the first item to be processed.

statement A meaningful expression that describes or specifies operations and is complete in the context of the BASIC programming language.

stop bit A signal following a character or block that prepares the receiving device to receive the next character or block.

storage A device, or part of a device, that can retain data. Memory.

string A sequence of characters.

subdirectory Any directory contained in the root directory list or within another subdirectory list.

subscript A number that identifies the position of an element in an array.

syntax The rules governing the structure of a language.

syntax error An incorrect instruction resulting from a misspelling, missing or faulty punctuation, a missing or incorrect character.

table An arrangement of data in rows and columns.

target In an assignment statement, the variable whose value is being set.

telecommunication Synonym for data communication.

terminal A device, usually equipped with a keyboard and display, capable of sending and receiving information.

toggle Pertaining to any device having two stable states; to switch back and forth between the two states.

trailing Located at the end of a string or number. For example, the number 1000 has three trailing zeros.

trap A set of conditions describing an event to be intercepted and the action to be taken after the interception.

tree-structured directory A group of related files and directories on the same disk organized in a hierarchical structure, as in a "family tree."

truncate To remove the ending elements from a string.

twos complement A form for representing negative numbers in the binary number system.

typematic key A key that repeats as long as you hold it down.

variable A quantity that can assume any of a given set of values.

vector In computer graphics, a directed line segment. More generally, an ordered set of numbers, and so, a one-dimensional array.

viewport In computer graphics, a defined area of the screen.

window In computer graphics, a defined area in the world coordinate system.

world coordinate system A coordinate system not bounded by any limits – unlimited "space" in graphics.

wraparound The process whereby parts of an object that are not visible within the window produce incorrectly drawn images due to the overflow of internal coordinates.

write To record data in a storage device or on a data medium.

zooming In computer graphics, causing an object to appear smaller or larger by moving the WINDOW and specifying various WINDOW sizes.

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z

The Personal Computer
Hardware Reference Library

Reader's Comment Form

BASIC Reference 6361134

Your comments assist us in improving the usefulness of our publication; they are an important part of the input used for revisions.

IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for technical questions regarding the IBM Personal Computer or programs for the IBM Personal Computer, or for requests for additional publications; this only delays the response. Instead, direct your inquiries or request to your authorized IBM Personal Computer dealer.

Comments:

Tape Please do not staple Tape

Fold here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 321 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432